REFERENCE

Message Passing - How Software Engineers Use Talk

Christopher David Bates

A thesis submitted in partial fulfillment of the requirements of
Sheffield Hallam University
for the degree of Doctor of Philosophy

October 22, 2014

Dedication

I would like to dedicate this thesis to my mother Margaret, to my late father George, to my wife Julie and our wonderful daughters Sophie and Faye.

Acknowledgements

I would like to acknowledge the support and help I've had from my supervisory team.

Dr. Kathy Doherty has been an inspirational guide into the world of ethnomethodology.

Dr. Karen Grainger has given invaluable help with the nuances and minutiae of conversation analysis.

My fieldwork took place at a number of companies in Sheffield and Nottingham. I am immensely grateful for the access which they gave me and for the time which individual members of staff spent with me.

Finally I'd like to acknowledge the contribution of Professor Simeon Yates at the outset of this project.

Abstract

In this thesis I present software engineering as a social process in which programmers work together to create technical solutions. I propose that the social structures which developers create and within which they work provide the foundations from which they are able to collaborate to build software. In doing so I characterise software engineering as being as much a social enterprise as it is a technical one.

Since its origins in the late 1960s, the discipline of software engineering has been one that is concerned primarily with tools, techniques and processes. Research and writing within the area, by both academics and practitioners, have been interested in developing better ways to deliver better software and, hence, better customer satisfaction. Relatively little effort has gone into understanding what it is that software developers, in particular programmers, do as they work collaboratively.

Starting from an ethnomethodological position, I present an examination of those activities of programmers which enable both sense-making and coordination. The research examines the work of developers in teams that self-identify as adherents to the Agile Manifesto. These teams are interesting because of the Manifesto's commitment to a social view of software development.

Three teams of professional developers are studied in their normal working environments as they work on commercial projects for their clients. The first is a failing team which I follow as they begin to use Scrum whilst the second team has been using Scrum for a number of years. The final team uses a mixture of techniques from XP, TDD and Kanban to create their own way of working.

By revealing the ethnomethods of developers across the three organisations I show that the design and implementation of code is enabled through social interaction.

Contents

Dedication i

Acknowledgements ii

Abstract iii

Contents iv

List of Figures 1

1 Introduction 1
- 1.1 Software Engineering 4
- 1.2 The structure of this thesis 7

I Background 9

2 Software Engineering Practices 10
- 2.1 Introduction 10
- 2.2 The difficulty of developing software 11
- 2.3 Structuring work as projects 16
- 2.4 Chandler: a project in crisis 20
- 2.5 Agile methods 24
  - 2.5.1 Extreme programming 25
  - 2.5.2 Scrum 28
- 2.6 The Agile Manifesto 37
- 2.7 Agile practices 40
- 2.8 Software Engineering as Craft 49
  - 2.8.1 Craftsmanship 50
- 2.9 Summary 52

3 Talking about Software Engineering 54

3.1 Introduction 54
3.2 Ethnography 55
3.3 Ethnomethodology 60
3.3.1 Conversation analysis 64
3.3.2 Face-work 68
3.4 Negotiating design 69
3.5 Finding meaning 73
3.6 The role of representation in talk about programming 76
3.7 Communication and coordination 80
3.7.1 Organising teamwork 82
3.7.2 Communicating within development teams 84
3.7.3 Discussing technical issues 87
3.7.4 Shared perspectives within teams 89
3.7.5 Being a community 91
3.8 Empirical research into software engineering 93
3.8.1 Criticisms of empirical software engineering 96
3.9 Ethnography and Software Engineering 98
3.10 Summary 101

4 Methodology 103
4.1 Introduction 103
4.2 Principles which underpin the design of the study 104
4.3 On doing ethnography 106
4.3.1 Fieldwork 106
4.3.2 Field notes 110
4.3.3 Writing ethnography 111
4.4 Implementing the study 113
4.4.1 Finding cases 115
4.4.2 Designing and doing the field work 117
4.4.3 Taking notes 122
4.4.4 Transcribing the recordings 124
4.4.5 Answering the questions 125
4.5 Ethical considerations 126
4.6 The three cases 128
4.6.1 Z * 131
4.6.2 A*,com 134
4.6.3 E * 137
4.7 Analysing the data 139
4.8 A mixed methods approach 142
4.9 Summary 144

II Case studies **146**

5 Z* **147**
5.1 Introduction 147
5.2 The company 149
5.2.1 DVD authoring software 149
5.3 Measuring success 152
5.4 Negotiating access 153
5.5 The first day 157
5.6 The staff 161
5.6.1 Project management 162
5.6.2 The developer 167
5.6.3 Quality auditing 171
5.6.4 The product managers 174
5.7 The first Scrum 176
5.8 The Daily Scrum Meeting 179
5.9 Discussion 181

6 A*.com **185**
6.1 Introduction 185
6.2 The company 186
6.2.1 The product 187
6.2.2 Approaches to development 189
6.2.3 The staff and their working environment 190
6.2.4 Technologies 192
6.3 Working practices 195
6.3.1 Being agile 196
6.3.2 Skype 198
6.4 The stand-up 199
6.5 Managing the meeting 201
6.5.1 Taking turns 201
6.5.2 Ending the meeting 207
6.5.3 Authority 208
6.6 Accounting 212
6.7 Humour 216
6.7.1 Successful humour 217
6.7.2 Unsuccessful humour 220
6.7.3 Banter 223
6.8 Discussion 226

7 E* **230**
7.1 Introduction 230

7.2 The company 233
7.3 The development team 236
7.3.1 Workflow 238
7.3.2 Coding 240
7.4 Gathering the data 244
7.5 Working with existing code 245
7.6 Talking about testing 253
7.7 Implementation or design? 263
7.8 Discussion 271

III Discussion **275**

8 Discussion 276
8.1 Introduction 276
8.2 Research themes 277
8.3 Contribution to knowledge 280
8.3.1 Negotiating sharedunderstanding 281
8.3.2 Coordinating work 286
8.3.3 The Agile Move 291
8.3.4 Qualitative softwareengineering research 295
8.4 Further work 296
8.5 Conclusions 299

IV Bibliography **303**

Bibliography 304

V Appendices **324**

A CA Symbols 325

B Website 326

C Consent form 335

D Stand-up at A* 337

E Existing code at E* 346

F Testing at E* 353

List of Figures

2.1 The waterfall model, (Boehm; 1988) 17
2.2 The spiral model, (Boehm; 1988) 18
2.3 Structure of Scrum 30

5.1 Old and new communication structures 163

7.1 The developers' organisational structure at E* 237
7.2 The database structure 248
7.3 The architecture of the system 251

Introduction

This thesis is a study of software engineers and software engineering. The research questions which are addressed here are:

- how do software teams coordinate their work through their talk-in-interaction about that work?
- how do software development teams which use agile methods create and sustain an agile culture?
- how do software engineers talk about code so as to make sense of it?

In anwering these questions this thesis will reify software engineering as an activity which is both social and technical.

Software engineering research has a strong focus on tools, languages and techniques but Harper et al. (2013) see this research as "noticeable for the lack of attention they give to the felt-experience of software engineers". Some academics and practitioners have begun to challenge purely *technocentric* conceptualisations of software development by foregrounding an alternate view of it as a social activity. Typically, this more social view of development has examined the relationship between developers and the end-users of software, trying to improve the gathering of requirements or the usability of software

applications. There has been far less interest in the social relationships between developers as they produce code.

Whilst academics in the 1990s examined communication with users, some practitioners were bringing the communications needs of developers to the fore. The agile methods movement takes as one of its core principles that improving communication within a team can improve the performance of that team. The Agile Manifesto, (Beck et al.; 2001), and the working practices which are used by agile teams present a challenge to established tropes of software engineering in their explicit appeal to developers. These methods privilege interactions within teams and the use of tools and techniques such as Test-Driven Development over documentation and project management.

There is evidence that agile approaches lead to a range of benefits from happier developers, (Muller and Padberg; 2004) and to more productive teams, (Benefield; 2008). Dybå and Dingsoyr (2008) write that it is not clear how agile methods are used in practice nor are their benefits and the causes of those benefits necessarily well-defined. In particular the role of communication is under-researched and under-theorized. Are agile developers more productive because they are placed in situations such as Daily Scrums in which they have to talk about their work or despite this? Does pair-programming lead to better code through talk-in-interaction about code?

The development of software happens in many places and at many times but this work is interested in professional programmers working on commercial projects. Llewellyn and Hindmarsh (2010) write that research into organisations rarely turns an analytical focus to questions of what work is and how work is accomplished. Research into software development seems rarely to re-

veal the working practices of developers, often being more interested in tools and techniques than in the work which they support. Schultze (2007) points out that much research into knowledge work has focused on the classification of knowledge rather than on the work of producing it which has meant an interest in what people know rather than in how they come to know.

This thesis grew from an interest in what programming *is* and in what programmers *do* when they program. Software development in general and programming in particular, are situated in "work" whether that is seen through the abstractions of project and team or seen as simply another type of white-collar office work. This research shows how programmers work together to create software and how the ways in which they talk about their work and the software which they are making frame and formulate that work.

Software engineering has a disciplinary orientation towards the tools, processes and products of software engineering and significant improvements have been made in all of these. But because the discipline's interest in process means that whilst there has been significant research into aspects of agile such as project management relatively few studies have been made which look at communication within agile teams. Those studies which do exist have tended to look at the management of agile teams or at the process of designing software. The work which is presented in this thesis is specifically and exclusively an examination of that part of programming which is the writing and testing of software source code. It addresses the ways in which agile developers talk about their work and how that talk, in turn, enables agile practices.

1.1 Software Engineering

The development of software is a multi-faceted activity which includes both the technical and the social. It is heterogeneous engineering, (Law; 1987), in which technologies arise through the negotiation of a complex network of social and technical factors mediated through conflict. Law is aware that engineers are as much products of a mix of factors as the artefacts which they create are. The engineer and client both bring complex ideas, experiences and requirements to the negotiation of a product. In a sense this has been known since the formulation of Conway's Law which states that "[a]ny organization which designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure", (Conway; 1968). Any examination of engineering practices which omits these wider contexts is, Law argues, in danger of over-simplifying the activity.

The IEEE has defined software engineering in (Abran et al.; 2004) as:

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

Software engineering and computer programming are different things yet even some professional programmers might struggle to identify what it is that differentiates software engineering and computer programming. The differences between the two terms might become clear after looking at some definitions. The terms work as effective synonyms in general usage because whilst

relatively few people have heard of *software engineers* everyone has heard of *computer programmers* even if they have no idea what a computer programmer actually does.

The Association of Computing Machinery write in (ACM; 2012) that:

> Software engineering is concerned with developing and maintaining software systems that behave reliably and efficiently, are affordable to develop and maintain, and satisfy all the requirements that customers have defined for them. It is important because of the impact of large, expensive software systems and the role of software in safety-critical applications. It integrates significant mathematics, computer science and practices whose origins are in engineering.

(Sommerville; 2004), a popular introduction to Software Engineering, widely used on University courses, states that:

> Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use.

Those definitions all include notions of:

- Disciplined processes.
- Lifecycles running from the specification of a system through development and on to maintenance.
- Engineering.

But the notion of *programming*, of writing and developing code is not a core part of them. Programming is implicit in terms such as "development", "maintenance" or "production", yet, surely, writing code *is* software engineering. The authors of those definitions place the design and creation of high-quality code as neither more nor less important than requirements gathering, project management or the writing of a specification.

The idea that software can be "engineered" is fundamental to contemporary understanding of the software development process. Professionals across IT take engineering to mean the application of rigour and of numerical analysis and the creation of controlled and reproducible processes, (Brechner; 2007). Engineers from other fields have more nuanced ideas of what engineering can mean. Florman (1976) writes that engineering is "the art or science of making practical application of the knowledge of pure sciences" and that engineers "solve problems of practical interest...by the process we call creative design". Engineering "has as its principle object not the given world but the world that engineers themselves create", (Petroski; 1992).

The larger thesis put forward by Petroski is that engineering is an essentially human activity and that engineered structures and products sometimes fail but that the important thing is to learn from those failures so that they are not endlessly reproduced. Software engineering is a discipline born of crisis, (Naur and Randell; 1968), one which see itself as still in crisis, (Glass; 2006b), and one which is ill at ease with itself, (Bryant; 2000), because of its failures. Both practice and research in software engineering are framed within an overarching need to reduce both the frequency and severity of systems failure. The rigour of engineering provides possible route from crisis to reliable software.

Yet software engineering's vision of engineering as rigorous and process driven differs from the engineer-authors who see it as a practical and creative activity in which there is continuous learning. Counter movements such as Agile, see Section 2.6, and Craftsmanship, Section 2.8, whilst not rejecting the software engineering agenda have started to re-formulate software development as something which is more akin to the design-led ideas of engineering.

1.2 The structure of this thesis

This thesis is divided into three parts.

In the first part, Chapter 2 introduces software engineering as a discipline in which the delivery of product can be problematic. A case study shows why producing software can be so difficult. The Chapter then examines alternative approaches to team working and looks at the role of the skills of individual developers in the production of software.

Chapter 3 studies the importance of talk as a method for the production of understanding and coordination. Finally in this part, Chapter 4 includes a discussion of relevant research methods and the design of the research used in this work.

The second part contains the three case studies which were undertaken. The first case study, Chapter 5 follows a software house as they introduce the agile method Scrum into a project. It shows their working practices before they started to use Scrum, discusses the problems managers in the company identified as drivers for the move to Scrum and, finally, shows how the team applied some of Scrum's methods.

The second case study is of a company which had been using Scrum for a

long time. In this Chapter a single event, one of their daily stand-up meetings, is analysed in detail. The analysis reveals how their social interactions within the meeting enable the coordination of work, the sharing of technical detail and strengthen the personal relationships within the team.

The final case study is a detailed analysis of the sense-making in which two developers are engaged as they work in a "pair" to understand existing code and write more code based upon it. This case study is presented as a conversation analysis of a series of interactions which took place in a period of three weeks.

The final part of the thesis contains a discussion Chapter in which the key ideas and themes of the thesis are brought together to demonstrate that the practices of software engineers as they try to understand their code and as they coordinate their work are socially situated. This discussion shows that, although software development is at its core a technical activity, the social context within which it happens affects how it is done and what it produces.

Software Engineering Practices

2.1 Introduction

This Chapter gives an introduction to the professional activities of software engineers, specifically those activities which help them when they write, test or understand code. The practices of software engineers are the main topic of interest within their discipline, a discipline which arose from a perceived need to identify and codify practices which helped improve the delivery and quality of software. Software engineering is a diverse activity which changes as technologies change and as our expectations of the capabilities of software and systems change.

In the last decade an important, and radical, change has revolutionised the working lives of many programmers. Low-cost, rapid, iterative and incremental approaches to development which place the skills and knowledge of programmers at the heart of the engineering of software have become mainstream. These *agile methods* are, in part, a response to a variety of types of project failure. But they may be as much a response to the changing needs of developers as they are to the problems faced by project managers or customers.

Communication is a core idea in the agile community and is central to the ways in which many agile practices are implemented. This Chapter introduces those agile practices which are concerned with communication but looks specifically at a subset which were found in the fieldwork for this thesis. Examining agile communication as seen in the commercial development of software and provides a context within which to read the later empirical and analytical Chapters.

2.2 The difficulty of developing software

We have been programming computers since the 1940s and for most of that time there has been talk of a crisis in software development. That crisis stems from the perception that most IT systems are delivered late, over budget and with less functionality than was originally specified. The situation became so bad that in 1968 NATO convened a conference of 50 experts to examine the situation and suggest a way forward. Much of the discussion of this event in Naur and Randell (1968) remains pertinent today, indeed we often repeat the same discussions with the technologies simply updated. This conference popularised the use of the term Software Engineering and set its basic parameters. The new discipline was based on study of methodologies, how programs are designed and written, and tool support for development.

The noticeable omission from both the 1968 conference, and most later work, is the idea of the software engineer as a person who designs and writes computer programs whilst working collaboratively and creatively. In part that has been deliberate: the discipline of software engineering was established with an engineering bent, d'Agapayeff was quoted as saying "[p]rogramming

is still too much of an artistic endeavour. We need a more substantial basis...". This view prevailed in 1968 and continues to dominate today. The idealised "software engineer" who rises from the Garmisch conference regards engineering as an activity which is essentially rigorous because the mathematical certainty of "engineering" brings repeatability and "measurability" to software development, (Brechner; 2007). By constituting the creation of software as a process which can be measured and managed the *process* became an objective reality. The formulation of this reality happened so quickly that by the time that NATO reconvened its conference in Rome in 1969, Brian Randall noticed that participants talked of "software engineering" as an established fact, (Rosenberg; 2007).

The 1968 NATO conference was convened because of widespread perceptions that there was a crisis in software development. At the time the most infamous example of a failed project was IBM's development of its OS/360 operating system. The OS/360 project had gone disastrously wrong: the system was delivered over a year late, required 5,000 person-years effort, cost $500 million (four times the original estimated cost) and was full of bugs which would take years to track down and fix. Projects like OS/360 were failures along both technical and managerial dimensions. Other projects were failing because the needs of users were not considered deeply enough so that time and money were being spent to build systems which weren't needed or which didn't perform as their users expected or required.

It was clear that software development, as constituted in 1968, was unable to keep pace with the changing needs of customers or with the pace of developments in hardware. Something had to change. Naur and Randell report

two of the Garmisch participants talking about the ways in which they saw systems being developed.

> Kinslow: The design process is an iterative one. I will tell you one thing which can go wrong with it if you are not in the laboratory. In my terms design consists of:
>
> - Flowchart until you think you understand the problem.
> - Write code until you realize that you don't.
> - Go back and re-do the flowchart.
> - Write some more code and iterate to what you feel is the correct solution.
>
> If you are in a large production project, trying to build a big system, you have a deadline to write the specifications and for someone else to write the code. Unless you have been through this before you unconsciously skip over some specifications, saying to yourself: I will fill that in later. You know you are going to iterate, so you don't do a complete job the first time. Unfortunately, what happens is that 200 people start writing code. Now you start through the second iteration, with a better understanding of the problem, and it is too late.

The problem of software development which Kinslow identifies stems from incomplete specification. The specification contains the requirements for the system. Those requirements come from an understanding of the problem domain and of the solution domain. However, that understanding only comes

gradually. Understanding requirements then designing software to implement a solution to those requirements necessarily has to be an iterative process. But systems are large and complex. In the 1960s software was intimately tied to hardware with operating system, application software and, sometimes, the hardware itself being built concurrently. Project teams quickly became large and their members became specialists in particular aspects of the solution. Once an approach had been determined and the team had started to implement it changing course was very difficult. Projects tended not to iterate, they tended to build the thing which they first defined.

Another participant agreed that requirements presented a problem.

> The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job.

Not everyone agrees that there really *is* a crisis. If there is a crisis in software development it "represents a damning condemnation of software practice. The picture it paints is of a field that cannot be relied upon to produce valid products", (Glass; 2006b). Yet we know that innumerable software applications ranging from Websites to operating systems are successfully developed and deployed. Glass blames a lack of studies looking at the industry's successes: "Many academic studies assert the software crisis is the reason behind the concept the particular study is advocating, a concept that is intended to address and perhaps solve this particular crisis".

The best way to understand the failure of projects may be to look at lots of them. Consultants DeMarco and Lister have studied and worked on many software projects over the last forty years. Every year from the late 1970s through to the end of the twentieth century they surveyed project teams about their results. In examining over five hundred projects they found "15 per cent...came to naught" because they were, for some reason, cancelled or never used. "25 percent of projects that lasted twenty-five work years or more failed to complete". For those failed projects the "overwhelming majority" had "not a single technological issue to explain the failure". The most common reason given for project failure was "politics". When they unpicked the idea of politics they found that the term was used to cover a host of people-related issues such as communication or staffing problems, difficulties with the client or a lack of motivation within the team, (DeMarco and Lister; 1999).

When projects fail because of organisational politics or culture, improving those things, or at least recognising that they have become problems and responding accordingly, ought to bring more successful development. Chapter 5 examines exactly this situation.

The development of software *is* difficult but so are many other types of engineering. Software is different because it is an ephemeral manifestation of ideas. It may be software's very lack of substance, its ability to become anything which is sufficiently different that its creation requires its own special crisis. "Software is different because of its intangibility and plasticity", (Diaz-Herrera; 2009). Perhaps because software is not concrete the processes which are used to develop it become as plastic as their product: "the great majority of software produced is developed following ad-hoc methods, and it remains

true today that software engineers have not been able to put together a coherent engineering design method for the systematic production of software, specifically for large, complex, ill-defined systems", (Diaz-Herrera; 2009).

An intangible product rather than a lack of rigour in its development might be the reason we have a software crisis. In some ways, perhaps, the software crisis exists because it is so widely talked about. It could be that "[o]ur basic problem is simply the success of modern computer science. History has shown that this truth is very hard to believe. Apparently we are trained to expect a *software crisis*, and to ascribe to software failures all the ills of society: the collapse of the dot.com bubble, the bankruptcy of Enron, and the millennial end of the world", (Noble and Biddle; 2002).

2.3 Structuring work as projects

Software is developed by teams of people whose work is generally structured as projects. Much of the discipline's focus is on improving the industry's approach to managing different types of project using a variety of structures. Organisations such as the Software Engineering Institute have developed models of best-practice for project teams and organisations which are widely followed, (Software Engineering Institute; 2010). Most people have some commonsense understanding of what a *project* is, but what is a software engineering project? How is the term understood by practising software developers?

The Software Engineering Institute defines a software engineering project as "a managed set of interrelated resources that delivers one or more products to a customer or end user. This set of resources has a definite beginning and end and typically operates according to a plan. Such a plan is frequently

documented and specifies the product to be delivered or implemented, the resources and funds used, the work to be done, and a schedule for doing the work. A project can be composed of projects", (CMMI Product Team; 2001). That definition is self-explanatory and is probably not too dissimilar to the commonsense definitions which most people would give.

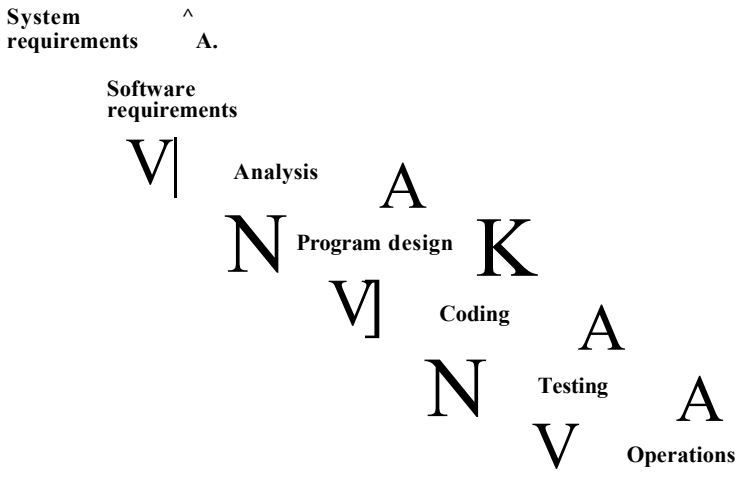


Figure 2.1: The waterfall model, (Boehm; 1988)

Although projects can be *ad-hoc* and free-form, the conventional approach to development is plan-driven, (Boehm and Turner; 2004). These projects move naturally through a number of distinct phases from initial requirements-gathering towards a deployed solution. This approach to software development has, in theory, a number of benefits for developers and their employers, not least that resourcing needs, costs and due dates can be agreed upon at the start of a project. The best known approach to plan-driven development is the water-fall model which is shown in figure 2.1.

In an idealised project which is following the waterfall model, all of the customers' requirements are specified at the start of a project and codified in

architectural documents, designs, models and implementation plans, (Sommerville; 2004). Of course reality is never this straightforward which is why the waterfall model is often criticised for its simplicity. (McBreen; 2001) writes that with waterfall "it is easy to consume half the available time before anything can be demonstrated to the users". The problem is that the project structure is heavily front-loaded so that effort is put into the definition of requirements and the design of the software rather than into implementation. Few projects are either so straightforward or so heavily resourced that a perfect set of documentation can be produced such that it will naturally lead on to development, testing and deployment. Over the years a number of variants have appeared which maintain a well-structured project but which allow for more flexibility during the lifetime of the project. The spiral model, for example, describes a more realistic project structure, Figure 2.2 shows how it works.

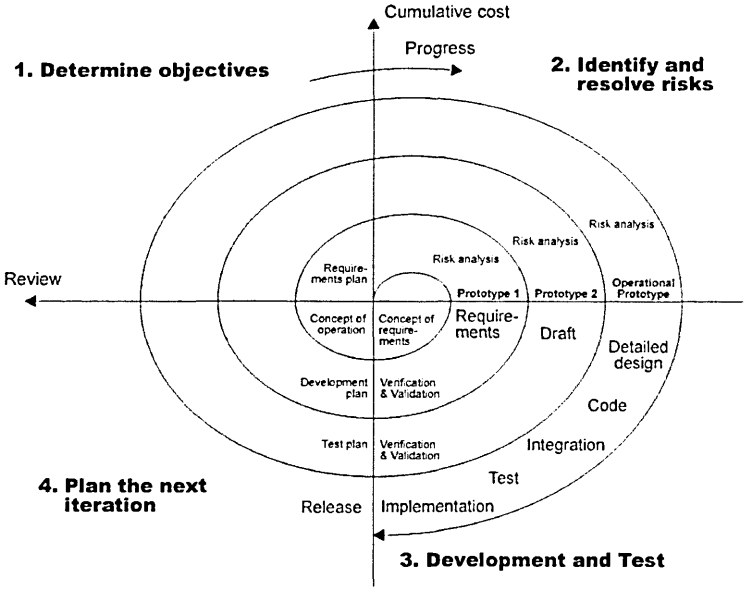


Figure 2.2: The spiral model, (Boehm; 1988)

The majority of software development has been done using, or attempt-

ing to use, a structured approach in which the structure and functionality of the product is understood relatively early in the development cycle. In a significant proportion of projects understanding of the functionality which will appear in the completed product is not developed until well into the development process and project structures can act to restrict understanding within teams, (Banker et al.; 1998). Iterative development is a successful answer to the problems of incomplete requirements and limited understanding of the necessary path to a completed implementation, (McBreen; 2001, Harper et al.; 2013).

In iterative development the team adds new functionality as and when it is needed. The project continually cycles through requirements definition,design and implementation phases. Usually the developers agree the functionality which they will implement in the next cycle with the customer, or their proxy, and determine the length of the cycle and the resources which are required to complete it, (Deemer et al.; 2010). Often the "new" requirements include the need to modify functionality which has already been implemented. Studies such as Hanssen et al. (2009) which look at changes made to code bases show that the same sections of code or the same parts of a product are repeatedly reworked. However repeatedly changing programs leads to "code entropy" in which the source code becomes so complex, or varies so much from its original form, that errors begin to appear within it. The developers then risk misunderstandings which "lead to a fear of changing the code, both for adding new features and for refactoring", (Hanssen et al.; 2009).

The need to constantly alter code is a common factor in three of the case studies presented in this dissertation. Chapter 6 is a study of a daily meeting

within a small team whose discussion focuses on difficulties each of them has with their code. Chapter 7 follows two programmers as they work together to understand some existing code as they write code which uses it. Chapter 5 follows a small software house as they introduce agile processes in response to their customers' ever-changing needs.

2.4 Chandler: a project in crisis

Rosenberg (2007) gives a detailed journalistic account of a long-running project to create an information management system called *Chandler*. The Chandler project was initiated and funded by Mitch Kapor who made his money from the creation of the seminal spreadsheet application *Lotus 1-2-3*. Kapor and his team wanted to build an application which competed with Microsoft's Exchange Server and Outlook email client but with some fundamental differences in technology and approach:

- Chandler was an Open Source project with a core team of paid developers and a wider volunteer community contributing code, bug fixes, design ideas and documentation.
- Chandler was to be built using a peer-to-peer architecture rather than a more conventional client-server one.
- The application was to be programmed in the language Python.

The project structure was similar to that used by others in the Apache Foundation, to some extent by the Linux Foundation, and by Mozilla as they created the Firefox browser. Each of these projects has a talented core team paid to design and implement the application which is supported by a com-

munity of volunteers. Often these developers are employees of companies such as IBM, Apple or RedHat who are paid to work on external Open Source projects. The team at Chandler was reasonably large, twenty seven developers at its peak, but never managed to build a supportive community effort. Instead almost all of the development work was done in-house.

The first prototype of Chandler was written by programmer Andy Hertzfeld using the platform-independent language Python. One of the benefits of Python is that it has a really simple syntax which means that programmers using it can be very productive, making it ideal for prototyping early version of applications. Python doesn't have its own GUI toolkit but there are bindings to an number of alternatives. Hertzfeld used a toolkit called WxWidgets, at the time called WxWindows, in his prototype. At the time of the Chandler prototype WxWidgets was still in heavy development with new features being added all the time and existing features changing in the way that they worked.

Because of the choice of technologies, building Chandler was always going to be a struggle. The peer-to-peer architecture was different to the architecture of other email and calendaring applications which use a centralised server. The changing GUI toolkit meant that Chandler's code was always in flux but that the rate of change was, in part, being determined by another project. Rosenberg gives many examples of the problems which this created for the team but one illustrates many of their difficulties.

The data held by each client needed to be held in a local database. Most projects would use a simple relational database – these are a well-understood, robust and relatively simple technology. The Chandler team decided to take a different approach called an object database. Specifically they would use a

product called ZODB which is written in Python. Learning to use such complex technologies immediately slowed development at a time when they were trying to produce the first release of their software.

Rosenberg describes a team meeting at which they argued about whether to ditch ZODB and instead write their own wrapper around Berkeley DB. Programming large systems often comes to a choice between re-using code and wiring together components or building things from the ground up. There is a constant debate between those who want to re-use and those who think software development is easier if you build just those parts which you need to complete your project. "Modularity, and interchangeable modular components are a key component of the modernist approach in software, as in architecture, marketing, production, and elsewhere", (Noble and Biddle; 2002).

This Utopian vision has been written about and discussed for at least twenty years but has never really come to fruition. This is because re-using code is hard, (Glass; 2001). Re-use in the small is easy, that problem was solved in the 1950s when programmers "built huge, useful collections of mathematical and data-processing library routines". We still do that in every software shop today. However when we try to re-use large amounts of code it "is and always has been an unsolved problem. In spite of the enthusiasm of the components crowd, finding in a library of components the precise one that will solve your problem at hand is nearly an impossible task". The major difficulty in re-using code is, Glass argues, "the variability in the problems we solve, and in the solutions we create".

Chandler ran continually late and over budget. During development the team was still evolving their ideas about what the product was going to be.

As they moved through beta versions new functionality was constantly being added to the application. Some of the new functionality had major impacts upon the architecture of the application. For example between release 0.3 and 0.4 they abandoned the peer-to-peer architecture in favour of one which supported the, then new, WebDAV standard for collaboration over HTTP connections. Whilst the WebDAV standard would become more widely supported at the time there were no Free implementations and the Chandler team found themselves having to write their own server.

Constant changes of functionality, implementation strategy or technology are enough to de-rail any project. Fred Brookes' inspiration for *The Mythical Man-month* was the failures he saw developing IBM's System/360 software. The team building Chandler knew about Brookes' work and about the many other failures of large development projects yet they repeated as many of those mistakes as they could. The Chandler project was doomed to failure from the start because neither its objectives nor its technologies were clearly defined.

The project continues today at `http://chandlerproject.org/`. The most recent version at the time of writing is 1.0.3 which was released in April 2009.

The fieldwork presented in this dissertation demonstrates some of the same failings that were found in the Chandler project. Technological change is a major factor in many software development efforts. Chapters 6 and 5 look at teams who have moved from custom frameworks to using ones which are more common. This is the inverse of the move Chandler made around ZODB but the drivers were similar: a desire for more control and for something which was a better fit to the current needs of the project.

2.5 Agile methods

Software development throws up many examples of failing projects. The question for the profession is how to address the causes of failure so as to improve both process and product.

The introduction of structured project and team management techniques through the 1980s and 1990s promised improved oversight and control of development teams. The efforts of the Software Engineering Institute, the professional bodies and others meant that by the end of the Twentieth century practitioners and academics knew how to manage a software development project. Yet projects such as the computerisation of patient records in the NHS continue to fail to this day. The complexity of projects and the size of the teams involved in implementing them is one reason that projects fail. Large projects cannot be specified accurately from the outset and it is, according to Brian Randall, "far better to employ evolutionary acquisition, i.e. to specify, implement, deploy and evaluate a sequence of ever more complete IT systems, in a process that was controlled by the stakeholders", (Flinders; 2011). James Martin, quoted by Flinders says "I've also seen 'best practice' lead to 'worst result' projects far too often and I believe that's the root cause of the problem: process has greater emphasis than outcome and that's not going to get a project over the line".

Those who manage developers have a wide range of techniques at their disposal, but when projects fail it is most often because of poor application of conventional project management, (Jurison; 1999, Jiang et al.; 2004). The "managerial side of the software development project, meanwhile, is often conducted without adequate planning, with poor understanding of the over-

all development process, and a lack of a well-established management framework", (Jiang et al.; 2004).

Developers knew all of this a long time ago: iterative and incremental development was being discussed as long ago as 1957, (Larman and Basili; 2003, Abbas et al.; 2008), and the 1968 NATO conference was, in part, an attempt to find ways of avoiding problems in project management. For whatever reason the "software crisis" continued and as it did so the search for alternatives grew. Abbas et al. (2008) list numerous studies which argued that linear models of development, and modifications such as Rational Unified Process and the V-model, were failing to address the realities of modern software development.

2.5.1 Extreme programming

In the early 1990s Kent Beck and his colleagues at Chrysler began to codify a way of working which came to be called Extreme Programming, (Beck; 2000). They were building a payroll system using traditional approaches. The project was failing so badly that two months before it was due to go into production it still didn't "compute the right answers". With the agreement of senior management the project started again using a small team and a radical approach to development. Using three week long iterations, user stories and a local domain expert the team was able to turn the project round.

Building on the Chrysler experience, Beck defined an approach to software development which he called Extreme Programming, usually more simply called XP. XP is a radical alternative to conventional software projects which, its proponents regard as rigid, hierarchical and driven by the needs of the de-

velopers' managers rather than by the requirements of either the developers themselves or their customers and end-users.

Extreme Programming is built from four values which are deliberately different to those which are normally found in business. A fully-functioning XP team places value on communication, simplicity, feedback and courage.

The XP values are not meant to be merely abstract concepts, they imply a common-sense understanding of four ideas. Communication is any type of communication between people who are involved in the project using any appropriate medium. Simplicity means to do the simplest thing possible. Feedback means that developers require immediate feedback on the state of the system which they are building, usually this comes from automated build and testing tools. Courage requires that individuals are empowered to speak out about the state of their own work and that of their colleagues.

Courage, feedback and simplicity are important values but they can only become meaningful to a team if members are able to communicate with each other. When members communicate they must be able to exchange both formal and informal information. Communicating like this requires trust, (Holmström et al.; 2006), and mutual respect, (Beck; 2000).

Team members should communicate openly and freely with each other and, in XP teams, they will often use a techniques such as pair programming as a focus for that communication. These techniques, allied to modern tool support, provide constant feedback about the project throughout the development phase so that problems can be identified and handled quickly. When there is regular and open communication, problems in the design or implementation can be raised, logged in bug-tracking software and prioritised

quickly.

Problems have less impact if the code and design are as simple as possible. Beck writes that "it is better to do a simple thing today and pay a little more tomorrow to change if it needs it than to do a more complex thing today that may never be used". It is easier to estimate the effort required to complete a simple task than to do so for a large, complex series of tasks. But learning to value simplicity is difficult for programmers who by training and inclination enjoy complexity. The final XP value is courage which means having the ability, as teams or as individuals, to do the right thing even if it is more difficult than not doing so. For example, work may need to be discarded even after a major investment in it.

Whilst the four values of XP are linked, they do not make a development approach on their own. XP builds a set of less abstract principles on top of ts values. The principles are implemented using XP's practices such as unit testing, pair-programming and small releases. All the values, principles and practices of XP combine into a flexible, iterative and cyclical approach to the development of software and systems. Chong (2005) found that the XP approach "provides a framework for standardizing the work of software development and making this work more effortlessly visible and accessible to members of a software development team".

In a longitudinal ethnographic study of an XP team by Sharp and Robinson (2006), working practices were seen to evolve from those of a waterfall team to suit the requirements of the XP approach:

> "**Daily rhythm** : Start of day $\rightarrow$ stand-up $\rightarrow$ pairing conversations
> $\rightarrow$ end of day

Rhythm of the iteration : Pre-planning → planning game → daily rhythm → retrospective

These rhythms are important to the team and their continued productivity. These rhythms are sometimes referred to as the heartbeat of XP".

In practice XP is an approach which might not suit every team. Sharp and Woodman noted that developers found pairing to be "intense and tiring" and that "... continual interactions between developers and the 'customer', i.e. the person who represents the business domain and end user of the product" could lead to "clashes of culture between the two roles". When pair programming developers can work unusually closely for long periods of time. This can become "like a marriage", (Robinson et al.; 2007).

The values and practices of XP, and other agile approaches, "are created and sustained by practice", (Sharp and Robinson; 2003), and may be "constitute a community of practice with mutual engagement, joint enterprise and a shared repertoire". Section 3.7.5 will discuss the creation of culture and community through shared ideas and experiences in detail.

2.5.2 Scrum

When most programmers think of Agile Methods today, what they envisage is likely to be Scrum. This method was developed by Jeff Sutherland and Ken Schwaber along with a large number of collaborators from industry and academia, (Sutherland and Schwaber; 2007).

The essence of agile processes is the ability to react to changing requirements. Each agile method has its own approach to the running of a software

development team but all of them place flexibility at the heart of their approach. Developers work through a number of core activities regardless of the process they use to create their software and manage their work. Instead of gathering all of the requirements of a project at the beginning before designing then testing and finally deploying the entire product, agile teams work on smaller pieces. An agile team will gather sufficient requirements for a short burst of work lasting from a couple of weeks to a few months. The project team will repeatedly run short cycles with new requirements driving each iteration through incremental change until the entire product has been deployed.

Proponents of iterative software development sometimes claim that it can only happen successfully if a number of conditions are met, (Rising and Janoff; 2000). These include placing individual developers and their skills at the heart of the process, collaborating with customers throughout development, treating working code as the major artefact of the project and reacting to change instead of blindly following a plan.

Scrum projects are organised around small closely knit teams working on rapid iterations of a develop-release cycle. Development takes place in tightly focused sprints which last just a few weeks. At the start of each sprint goals are set including the functionality to be included in the next release and a hard deadline is set at which time the software should be released. During the sprint, brief daily meetings are held to ensure that the team remains on-track.

Sutherland began to evolve Scrum at Easel Corporation in 1993. Working with Schwaber he codified the approach at OOPSLA '95. Figure 2.3 shows the structure of a generic Scrum. A product backlog, which contains all work that needs to be done on the product, is created and continually updated. A

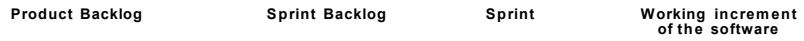


Figure 2.3: Structure of Scrum

subset of the backlog is selected to be worked on in the next iteration, aka *sprint,* which typically lasts for three weeks. Within the sprint their is a cyclical structure of daily meetings and at the end of the sprint a new version of the software may be released.

How Scrum works

The rationale behind, and use of, Scrum are thoroughly documented in Schwaber (1995), Rising and Janoff (2000), Sutherland and Schwaber (2007), Deemer et al. (2010) amongst many others. It is worth discussing briefly the key parts of the method because some of them will be interrogated in detail in later empirical Chapters.

Scrum divides the development process into a series of short iterations called *sprints.* Each sprint lasts approximately one month and includes major product development steps such as the refinement of requirements, design, implementation, testing and deployment. A sprint is *time-boxed* with a due date upon which it ends even if the scheduled work is incomplete. Scrum

includes a number of roles which should be filled. The key ones are *Product Owner*, *Team* and *Scrum Master*. Analogues of these roles don't really exist in a waterfall project where there will be layers of management, developers and customers with whom there isn't usually a structured relationship.

The Product Owner helps identify those features of the product which need to be worked on, assigns a value to each and helps prioritise them. The Product Owner sits outside the Scrum team but interacts frequently with them. The Team is the group of developers who are working on the project. Teams are typically around seven members strong. Ideally each member will be multi-skilled and the Team should be, or become, capable of self-organisation so that interference from external management is kept to a minimum. The key role is the Scrum Master who works with the Team and Product Owner to achieve their goals but, crucially, is not their manager. Scrum Masters facilitate the development and act as a bulwark against external forces which may move the Team off track.

Scrum is structured around a number of meetings. The main ones being the daily scrums, planning meetings and retrospectives.

Daily Scrum Meetings are probably the method's best-known feature. Each day the development team gathers for a short meeting and discusses their work. The meeting is lead by the Scrum Master. Each team member is asked three questions: what have they done since the last meeting; what are they planning to do next; and what impediments they face.

The Daily Scrum Meeting is not designed as a management meeting but as a co-ordination point – there are no long discussions and contributions from the team are kept to a minimum when each of them speaks. When team

members find impediments those are noted so that they can be solved outside the meeting often through discussion with the Scrum Master. The stand-up gives the team "the context in which expertise can emerge through interaction", (Faraj and Sproull; 2000). "Recognizing when and where expertise is needed is at the heart of heedful interrelating" according to Faraj and Sproull. The meeting demonstrates to the whole team where information, expertise or help is needed – these can be provided once the meeting concludes.

Daily Scrum Meetings have the potential to be socially awkward situations, particularly when there are difficulties with the development. Alternatively they can work to bond a team together through success or adversity. Chapter 6 examines the detail of one such event.

Each Sprint is defined at a planning meeting which is lead by the Product Owner. The team works with a Product Backlog, a list of tasks which need to be completed, to decided how much work they can achieve during the next sprint. The backlog is an organic list which grows and shrinks as the project progresses, often it is linked to bug-tracking software and includes some notion of priority, or relative importance, of the tasks. Teams choose tasks for a Sprint based on those factors which they feel matter most. These may include the customers' needs, the priority assigned to bugs or the amount of work which they can manage in a single iteration. The amount of work they can complete is estimated based on person-hours or on *story points*.

The third Scrum ceremony is the Sprint Review which happens at the end of each sprint. This meeting examines the team's achievements by looking at what they managed to deliver during the sprint. "A key idea in Scrum is inspect and adapt. To see and learn what is going on and then evolve based

on feedback, in repeating cycles", (Deemer et al.; 2010). The review is an opportunity to learn whether one's estimates were accurate and for the Scrum Master to determine if the work is "done". Anything which is not deemed to be done goes back into the backlog from where it may be selected by the Product Owner for the next iteration.

The final ceremony used in Scrum is the Retrospective. These are not always used but when they are undertaken retrospectives give the team a chance to examine and modify their *process*.

Research into Scrum

A number of authors have written about the use of Scrum in their own organisations. Kniberg (2007) provides a details discussion of his experiences using Scrum from managing the team to arranging the room in which they work. His book was written to help beginners, "[o]ne of the most valuable sources of information, however, was actual war stories. The war stories turn Principles and Practices into... well... How Do You Actually Do It". Pikkarainen et al. (2012) surveyed developers and managers and discovered the importance of management buy-in and of tailoring Scrum to the needs of specific teams. The idea that Scrum should be both adaptable and adapted has influenced a change in terminology from one of its originators who once wrote of *process*, (Schwaber; 1995), but now writes of *framework*, (Sutherland and Schwaber; 2007).

The question of how Scrum works out in practice is addressed by Benefield (2008) who examines the roll out of Scrum at Yahoo. Benefield joined Yahoo in 2005 to help them adopt agile methods. She found that they "started with

Scrum, using its lightweight framework to create highly collaborative self-organising teams that could effectively deliver better products faster". The reasons given for Yahoo's choice of agile methods which are expressed here provide some possible metrics against which the success of the approach might be measured: collaboration; self-organisation; product quality; and speed of delivery. These metrics are, themselves, difficult to define and measure because they are subjective, but through carefully constructed questionnaires or interviews some meaningful data can be generated.

Perhaps surprisingly Yahoo did not have an engineering process until 2002 when they implemented a "globally mandated waterfall process called the Product Development Process (PDP). Unfortunately, many teams simply ignored the process or, where they couldn't ignore it, paid lip service and made it look like they had adhered to the steps retroactively. The teams that did follow the PDP found that it was heavy, slowed them down, and added little real value". A number of teams had run unofficial agile projects but the first formal trial of Scrum at Yahoo took place early in 2005. Teams which took part in the trial had to commit:

1. to complete comprehensive Scrum training (which translated into Certified Scrum Master training for most members of the team);
2. to work with outside Scrum coaches during the first several Sprints;
3. to use all the standard Scrum practices described in Ken Schwaber's "Agile Project Management with Scrum"; and
4. to complete at least one Sprint.

Teams were free to leave the trial once their formal commitment to a single sprint had been met. Yahoo provided consultants, including Benefield, plus

coaching and training from agile leaders including Ken Schwaber, Paul Hodgetts, Mike Cohn, and Esther Derby. The organisational infrastructure around the Scrum teams was modified to remove impediments: "working with facilities to secure meeting rooms and take down cube walls, removing governance gates where processes were overly bureaucratic, and changing the way we conducted resource planning and portfolio management".

At the end of the trial period reactions were overwhelmingly positive. Benefield surveyed those involved and found:

- 74% saw some improvement in productivity across a 30-day period.
- 80% said their team's goals were clearer.
- 89% said that collaboration and communication improved within their team.
- 64% felt their team produced more value.
- 68% saw a reduction in waste.
- 77% felt positively toward scrum at the end of the trial.

Only one measure was more evenly balanced. 54% of those surveyed felt that the quality of their product improved during the experiment. 5% thought quality was reduced with the remainder feeling that the new way of working made no difference to the quality of their outputs.

Rising and Janoff (2000) report on their experience using Scrum with small teams at *AG Communications Systems*, AGCS, who build and install telecommunications infrastructure. AGCS were faced with the common problem of ill-defined or changing requirements. Developers would say "[m]ake the chaos go away! Give us better requirements!" but changing requirements are a feature of modern developments and companies such as AGCS need to find a

way to live with this reality. Rising and Janoff talked to the more successful development teams at AGCS and found that reasons they gave for their success included:

- We did the first piece and then re-estimated – learn as you go!
- We held a short, daily meeting. Only those who had a need attended.
- The requirements document was high-level and open to interpretation, but we could always meet with the systems engineer when we needed help.

These indicators look like a high-level view of Scrum. The match between practices which worked for some of their teams and an established software development methodology encouraged AGCS to experiment with Scrum. Three teams, working on different products, trialled Scrum. Each team found its own way of using Scrum, and whilst none of them followed all of the practices religiously, all found benefits from the approach.

Daily Scrum Meetings saw "the team began to grow together and display increasing involvement in and delight with others' successes". Problems moved from being there for individuals to being owned by the whole team. "Because the team is working together toward a shared goal, every team member must cooperate to reach that goal. The entire team immediately owns any one individual's problems". Achieving this idealised team ownership of work and, especially, the artefacts which are produced, and creating an "egoless team", (Weinberg; 1999) is difficult. Individual egos can easily inhibit progress, (Doershuck; 2004). Even when colleagues share openly and freely they may do so for egotistical reasons rather than for altruistic ones,

(Perlow and Weeks; 2002). Even when the team structure is a flattened hierarchy and the team runs as a democracy it can still fail. Jurison (1999) reports that egoless teams are ineffective in software development "because people, particularly highly talented software developers, do have egos". It is not clear that agile methods have solved this problem, there is talk of shared ownership and egoless teams but the reality within teams may be different. This area is currently under-researched, Chapter 6 shows how ego and personality impact upon the conduct of a daily stand-up in one Scrum team.

At AGCS one team suffered from changing requirements and a change in the organisational context for their product. Daily meetings and the use of a backlog, held in a spreadsheet, helped them prioritise their work because in "the daily meetings, the Scrum Master would call attention to backlog-item priority". Another team had a heavy testing schedule but found that regular meetings were an efficient way to share information so that "[t]he group as a whole decided the kind of testing to perform in the next test time, not just the tester who worked that test time".

Rising and Janoff conclude that although Scrum's practices are not new, it is basically "incremental time-boxed development" with added daily meetings, Scrum is appropriate for those projects which have ill-defined requirements or which exist within chaotic and changing conditions.

2.6 The Agile Manifesto

The originators of a number of agile methods and advanced software development techniques came together at the turn of the Century to write a statement expressing their core shared values. This statement became *The Agile*

Manifesto, (Beck et al.; 2001), and was aimed at the whole industry as. The Manifesto has four values which are based on twelve principles. It states that signatories value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

Each of these values can be applied to any agile method and most teams who use agile methods would agree with all four statements. The first and third values express the importance of the people on a project, including customers and end-users; the second and fourth values have more technical foci. The Manifesto has had an impact which is far wider than its own values. It has created a new culture of software development which modifies the traditional engineering approach through the deconstruction of monolithic projects into small iterations.

All of the seventeen people who co-wrote the Manifesto came from similar development communities based around the programming language Smalltalk, early work on object-oriented development and software design patterns, (Hazzan et al.; 2010). Their backgrounds meant that they were used to using both iteration and collaboration. One said "Smalltalk was so interactive that we just said, well, let's just program it and see", another respondent thought "some of the influence [on agile development] was object-oriented design, when people [used... incremental development, rather than trying to figure out [all the requirements a priori]".

It is often assumed that the Agile Manifesto grew out of the failures of the software industry. Hazzan et al show that, at least in part, "technological forces fostered cultural changes". The Agile Manifesto came at a time when many developers were working on smaller Web-based projects, when time-to-market was reduced from months or years to periods measured in *internet time*. Developers were beginning to have more interaction with customers and those customers often knew more about what they wanted from their systems than had been true ten years before. The Manifesto has been signed by hundreds of people. It clearly spoke, and speaks, to many working programmers. The values of the Agile Manifesto appeal to developers who work within the context of changing requirements, short development timescales and multi-disciplinary teams. Part of its appeal is that it captures the daily realities of the working lives of many modern programmers.

Not everyone thinks Agile is the solution to problems of software development. Rakitin (2001) wrote "I've been waiting a long time for software engineering to become a respected engineering discipline". Rakitin characterises developers as either *hackers* or *engineers*. Hackers "talk to people when they are stuck, since they often prefer to work without specification" and who simply want to produce "something which works". Software development, Rakitin thinks, "will change for the better only when customers refuse to pay for software that doesn't do what they contracted for". Perhaps ironically this is also the view of the creators of agile methods who want to maximise value for their customers but who feel that excessive project management and rigid structures get in the way of achieving that aim.

Recent discussions on forums and email lists show that tensions between project managers and agile developers remain today. Project managers fear that agile methods empower developers who do not understand the "complex realities of organisational change management, governance, financial controls and longer horizon planning activities". Whilst agile developers think that status and ego play a more significant role: "the Project Manager is the project's hero and the Plan is sound (reality is wrong - when it doesn't conform, corrective action is taken). In agile, the project manager is a team member like any other, and reality always trumps the plan (so when they don't match, the plans are revised)", (Elssamadisy; 2010).

2.7 Agile practices

All agile methods share some common features based on the Manifesto's core values. They structure projects through iteration, they strongly emphasise the need to communicate with the customer and within the development team and they value the delivery of working software most highly. Each agile method takes its own approach to meeting the goals of the Agile Manifesto through the selection of a set of practices. Developers can choose from a wide range of practices: the methods tend not to be rigorous or prescriptive. Thus one can find Scrum teams using Kanban boards or XP projects without on-site customers or with optional pair-programming. Any agile practice may be adopted by any team because developers believe that they "have the experience needed to define and adapt their processes appropriately", (Turk and France; 2002).

Whichever practices are chosen the day-to-day lived experiences of agile

developers are largely the same as those of developers on non-agile projects. However, those experiences are found within structures which are created to enable software development rather than to simplify project management. Such experiences will be revealed by the developers' talk in the case studies in this work.

About Daily Scrum Meetings

The Daily Scrum Meeting provides a space for co-ordination and communication across the whole Scrum team. Agile approaches tend to share the belief that "a team can be more effective if

- the cost of moving or sharing information is reduced
- the time between making a decision and seeing its effects is reduced", (Cockburn and Highsmith; 2001).

Developers talk about their work in front of their colleagues in relatively general terms. The Scrum Master should work to avoid the meeting becoming becalmed in technical detail whilst letting each developer share information about their work.

The rituals of Scrum all meet Cockburn and Highsmith goals but the daily stand-up is especially important because it is lightweight and information is shared quickly and easily with those who need it. Instead of emailing the rest of the team with details of changes they have made or problems they have found, each developer can tell them directly at the meeting. The entire team knows where the problems are, how they might be solved and where changes must be made to the code base.

In the meeting's context "communication" is being conceptualised as the sharing of information. It tends to be a one-way channel in which the developer explains their work to the Scrum Master with no expectation of further detailed discussion. Other team members who are present at the meeting receive the information as a side-effect. Whitworth and Biddle (2007) found that team meetings "provide high levels of social accountability and support, and awareness of activity in a project, such as provided by information radiators, was seen to increase feelings of security and control in the team environment"

One of the side-effects of a stand-up meeting is that the shared information leads naturally to other talk outside the meeting. Because each of the developers knows what their colleagues are working on they are able to talk about to each other about the work of the entire team. Nothing is hidden from colleagues..

Teams do not have to adopt Scrum or use stand-ups to gain some of these benefits. Sawyer et al. (1997) examined a company in which teams were given access to a computer-supported team room. Teams would use the room to examine and discuss designs and code. Use of the room meant that group discussions became easier, "[t]he direct effect is to make it easier for developers to work together; enabling the production aspects. So, software development improvements at this site have emerged without increased engineering. Rather, they have emerged due to increased discussion".

Distributed meetings

In a globalised industry such as software development teams are increasingly dispersed across locations, often in different time zones. Many software de-

velopers have experienced working in multinational and multicultural teams as sections of projects are outsourced and offshored. The day-to-day management of these projects becomes more complex because of the team's distribution. Holding meetings is difficult when members are at multiple sites and when the morning meeting at one site is actually the mid-afternoon meeting at another.

Successful collaboration relies upon the sharing of knowledge and, more importantly, the creation of social relationships, (Layman et al.; 2006). Members of successful teams trust each other but trust is "more likely to be built if personal contact, frequent interactions and socializing between teams and individuals are facilitated", (Kotlarsky and Oshri; 2005). Distribution of teams "challenge[s] project processes such as communication, coordination, and control", (Holmström et al.; 2006).

An important, but often unacknowledged, problem for developers on distributed projects is simply knowing who talk to at other sites. Holmström et al note that staff on distributed projects have the same requirements of each other as if they were physically together but that servicing those needs can be difficult: "[a]lthough the need for informal conversation is extensive, people find it far more difficult to identify distant colleagues and communicate effectively with them".

Large teams, distributed teams or sub-contracted teams always have communication difficulties. Herbsleb and Mockus (2003) found that communication and co-ordination always provide difficulties in the development of software. Distributed working adds new difficulties which can severely slow the development process, in part because the amount of communication be-

tween team members drops off sharply as distance between them increases. Engineers with offices just thirty metres apart communicate as infrequently as those who work twenty-seven miles apart. "In organizations with rapidly changing environments and unstable projects, informal communication is particularly important", a situation for which agile approaches are designed. Informal communication is especially important to developers. In distributed teams "[t]he nearly complete lack of informal, *water cooler* conversation appears to have the consequence that people know much less about what distant colleagues are doing, who has expertise in what area, what the current status of plans is, and so on. In general, it seems that there is relatively little understanding of the overall *context* or background information at distant sites", (Herbsleb and Mockus; 2003).

Communication builds relationships. Where team members have strong relationships they trust each other more and are able to work together more effectively, (Kotlarsky and Oshri; 2005). Good communications leads to shared understanding so that "when team members have familiarity with their application domain and shared knowledge of the task and each other they are more coordinated and perform better", (Espinosa et al.; 2002).

Communication difficulties in distributed teams are not just caused by distance and cannot easily be solved using IT systems such as video conferencing. The distance between team members may be physical, temporal or cultural but the main problem in today's multinational teams is often language. One of the respondents in Holmström et al. (2006) says "[w]hen you have language difficulties initially causing confusion, I think cultural differences can actually drive further awkward situations, and it snowballs". These problems can be

avoided when their is greater *rapport* between the members of the team but "[l]ittle is known about creating rapport between globally distributed teams", (Kotlarsky and Oshri; 2005).

There are many examples of successful distributed projects. Sutherland et al. (2006) report on a Scrum-based project which was distributed between teams in the USA and USSR. Distributed Scrum teams usually follow a best-practice in which each site runs daily stand-ups which are brought together for a "scrum of scrums" which act a synchronisation point between sites. Sutherland et al. (2006) saw developers who, whilst physically located at a particular site, could be on any of the project's teams. In effect each team was distributed which increased the managerial effort required to run the project. The developers could work in any combination which helped them acquire collective knowledge of the project. Collective knowledge is "elements of knowledge that are common to all members of an organization... Collective knowledge is defined as a knowledge of the unspoken, of the invisible structure of a situation, a certain wisdom", (Kotlarsky and Oshri; 2005).

When staff are co-located informal talk such as *war-stories* happens at the water cooler or in the lunch room. In the days of punch-cards these stories were shared whilst waiting for the machine to load and run a program, (Weinberg; 1998). Sometimes workers simply have to make space to meet and share stories. (Orr; 1996) describes the working lives of photocopier repair technicians who spend their working lives on the road or at customer sites, rarely visiting their own company's headquarters. These technicians gather daily at a diner for breakfast when they talk about their work, their customers and the copiers which they maintain. Informal talk happens when people are physi-

cally close to each other and collaboration and co-operation arise directly from that informal talk, (Kraut et al.; 1988).

Distributed teams rarely have the chance to replicate the ad-hoc sharing of programmers waiting for punch cards of copier technicians waiting for pancakes. Instead their opportunities for talk tend to be limited to email, chat or online meetings. Layman et al. (2006) show that simple information-sharing strategies such as responding immediately to emails or supplying continuous access to data about processes and products can help form a united team.

Pair programming

Pair programming is one of the defining practices of Extreme Programming, see Section 2.5.1. In pair programming "two people program with one keyboard, one mouse and one monitor", (Beck; 2000). Because the two programmers only have one machine they must divide their work so that both think about the code which they are creating but only one types the code and controls the PC. The person doing the typing is sometimes called the *driver*, the other member is then called the *navigator*. The navigator must think about the problem and the solution and try to contribute ideas and alternatives.

Working successfully as a "pair" requires active participation from both programmers through continual discussion of the work. The navigator cannot simply relax until it is their turn to type but must work with the driver to produce a joint solution. Navigators do not "manage" the driver, rather the intellectual work of programming is divided between them, (Bryant et al.; 2008). Although thinking and designing are shared, (Plonka et al.; 2011) show that the division of labour between driving and navigating is not evenly split

across the pair: some individuals drive more than others. Additionally, the effort of a pair is not totally focussed on the computer because they "spend on average a third of the session without any computer interaction focusing mainly on communication", (Plonka et al.; 2011).

Using pair programming requires a commitment from both the programmers and their managers. Developers tend to change partners frequently, sometimes twice each day which means they all become familiar with the whole of the codebase. It also means that any piece of code that a developer writes may well be read by all of their colleagues at some point. This has a tendency to encourage simpler, more readable code which is self-documenting and strongly encourages the use of good test cases. Weaker programmers build skills more quickly in a pairing environment because they are exposed to the knowledge, ideas and problem-solving strategies of all of their colleagues. Sometimes new or inexperienced team members struggle because they do not know who to approach for help with their problems. Studies have shown "that knowing where expertise resided in their teams had a positive effect on performance", (Espinosa et al.; 2002).

When developers pair, especially when they switch partners regularly, expertise spreads across the team. This might seem obvious, that the weaker programmer will learn from the stronger but within a pair a "significant amount of talk is at an intermediate level of abstraction", (Plonka et al.; 2011).

Pair programming would have benefits such as increased team cohesion, wider understanding, even if it did not lead to better quality code. Beck writes that pairs are "more productive than dividing the work... and integrating the results". One might assume that a pair would produce less work than two

developers working independently but that seems not to be the case. Pairs, Beck argues, produce better code which requires less re-working and which has lower life-cycle maintenance costs.

Hannay et al. (2009) looked at a number of studies of pair-programming to examine whether common-sense understandings, such as Beck's, of the approach stand up to rigorous enquiry. Studies tend to look at simple measures such as the productivity of paired programmers, the time they take to complete task or the quality of the code which they write. Hannay et al. performed a meta-analysis of a number of studies which shows that the measurable benefits of pair-programming are contestable. There is evidence that better solutions are created for complex problems and that simple tasks are solved more quickly. However, the former requires much more effort and the latter tends to lower the quality of the solution.

More rigorous studies are needed before we can say that we really understand pair programming. "Only by understanding what makes pairs work and what makes them less efficient can we take steps to provide beneficial work conditions, to avoid detrimental conditions, and to avoid pairing altogether when conditions are detrimental", (Dybå et al.; 2007).

Many programmers enjoy working in a pair. Some choose to do so even when the approach is not mandated by their employers. Because pairing is not statistically more productive than working alone this seems strange. A programmer in a pair loses ownership of their code and has to work at someone else's pace. A common-sense view of work would suggest that both of these are undesirable attributes of a working practice. If developers are choosing pairing then there must be other benefits which outweigh any costs. Muller

and Padberg (2004) found that a "feelgood factor" exists for pairs. The performance of a pair does not correlate with their experience but does correlate with their work satisfaction which is greater for developers working in pairs. Some people like working with others and it might be that a particular personality type is best suited to pairing. The increase in feelgood which Muller and Padberg report might be a product of personality type. But other studies show that personality type does not correlate with productivity and is far less significant than the developers' experience or the complexity of the task, (Hannay et al.; 2010).

The studies by Muller and Padberg and Hannay et al. show that research into the impact of pair programming is still incomplete and slightly contradictory. These studies show that developers enjoy pairing, that pairs handle complex tasks better than individuals do and that these benefits are nothing to do with personality type. In Chapter 7 pair programming is examined using Conversation Analysis to reveal how the developers' talk-in-interaction affects their development of software.

2.8 Software Engineering as Craft

The idea that software is, or can be, engineered has largely been accepted by the community. Ideas of engineering lie at the heart of widely used metrics such as the Capability Maturity Model. Some developers have a rather different view of their discipline. The Software Craftsmanship movement looks backwards to notions of craft, skill and trades guilds to suggest that our current approaches to the development of software might be flawed in a number of ways.

2.8.1 Craftsmanship

The idea that we ought to be engineering software grew out of the perceived failures of major development projects through the 1960s. Software engineering brought rigour, control and management to projects but those same types of project continue to fail to this day. Some developers have set out an alternative which is based on the idea that developers with high levels of skill and commitment will produce better software.

The engineering view has prevailed and continues to dominate but an alternative one which places people at the heart of the development process does exist. This may range from structuring development as an intellectual activity within the workplace as DeMarco and Lister (1999) do, to seeing it as a craft activity as described by McBreen (2001), Glass (2006a), Sennett (2008).

The debate between craftsmen and engineers is far from settled. Brechner (2007) makes a strong case for both agility and engineering. Brechner's view is that you "craft a desk, engineer a car" and that the process of building software should be consistent and measurable. As evidence for this he cites a study in which he and other developers at Microsoft measured themselves writing code. Not surprisingly when solving a problem under experimental conditions, experienced developers wrote similar amounts of code and took similar time to complete the task. Brechner uses this as evidence that, whilst the software they build differs, developers are consistent and measurable. But it doesn't follow that because something can be measured under test conditions it can also be measured in real work with customer's requirements.

The craft movement takes its lead from the Agile Manifesto in viewing software development as a fundamentally human activity. It is something which

is undertaken by skilled people who need, wherever possible to maintain their skills through their professional practice. This is expressed most succinctly in the Manifesto for Software Craftsmanship (McBreen; 2001). For craftsmen, engineering is not a bad thing. Tool support is necessary but must never distract from the skills and knowledge of programmers working in teams. A programmer learns from, and teaches, her colleagues and is able to move from apprentice through journeyman to become a master craftsman[1]. This journey is one of learning through talking and doing which sees the developer moving between teams, organisations and approaches. Ultimately, methodologies and project management should be there to help us build better software not to help us be better at following methodologies or at being managed.

Building software requires that the developer be fluent with the rules of the programming language, more than competent with a range of tools and able to understand the language and nuances of the problem domain. Software development is an expression of bricolage, Levi-Strauss (1966), in which developers must be neither jack-of-all-trades handymen or experts at a single task but skilled craftspeople who are masters of their trade. The traditional project following a waterfall model sees development split into specialised phases such as requirements gathering, design, coding or testing. Each uses different people who have become skilled in just one thing and who lack the knowledge and experience to see projects holistically.

Software craftsmen do not have to work in an agile way. Most professional developers care about skills, personal development and the impact of their work on clients and colleagues. However, the craft movement foregrounds

[1]The gender-specific terminology comes through the Software Craftsmanship movement paying a knowing homage to medieval craft guilds rather than through sexism.

ideas such as Test-Driven Development, iteration and continual integration. These are the very practices to which Agile developers have turned in recent years. The relationship between Agile and Craft is very close because of the fundamental overlaps between their philosophies. This relationship mirrors the one between the object-orientation and design patterns communities and the early Agile thinkers in the mid to late 1990s. All of these groups place developers and the code which they produce at the core of their thinking about software. Many of the practices which are shown in empirical Chapters 5, 6 and 7 can be found in McBreen (2001), Hoover and Oshineye (2010) and other texts on craft.

2.9 Summary

Software is developed through a complex and diverse set of activities. There isn't a single standardised process which developers have to follow. As one reads and listens to practitioners and academics it can seem that there are as many approaches to development as there are developers. Some ideas have come to be commonly accepted throughout the industry and within the academy. Software development is seen to be in permanent crisis, software ought to be engineered like other products, documentation is always going to be useful and so on. All of these ideas are contested but, as this Chapter has shown, their roots are found in the community's response to the failing projects of the 1960s.

By the 1990s Software Engineering was being formalised and, concurrently, once radical ideas such as object-orientation were gaining mainstream acceptance. New ideas in project management and an early interest in agility came

from the same root: the desire to deliver working software on time and to budget. In many ways the Agile Manifesto and Capability Maturity Model Integration may be orthogonal ideas but each is a logical endpoint of the debates of the 1990s.

Throughout this Chapter a number of Agile methods and practices have been presented. They were selected not because they are the most widely used practices but because they are the ones which were found in the fieldwork. Each of them is a poster child for a different aspect of agile. Pair programming is the signature activity in Extreme Programming, Scrum is synonymous with the stand-up meeting and estimation underpins both XP, Scrum and newer, lightweight approaches such as Kanban.

Those activities are used Agile projects to support project management, coding and design but all have one thing in common. If they are to be used successfully they require that the developers, managers and customers talk to each other. In the next Chapter the role of talk as a work practice, and as a way of configuring work, is examined.

Talking about Software Engineering 3

3.1 Introduction

Chapter 2 discussed the origins of the discipline of Software Engineering, the recent appearance of agile methods and some practices of agile developers which are pertinent to this thesis. Software development was positioned as both a matter of technical competence and practice and as a social activity in which groups work collaboratively to design, test and build applications. If we are to understand software development then we must necessarily understand both the technical and professional competencies of software developers and software engineering as a social process.

One way to understanding the social processes within which software is constructed work is to observe that construction as it happens. Ethnography is a well-established and widely used approach to the gathering of data about cultures and to analysing and interpreting those data. In this Chapter the use of ethnography in studying workplace cultures is discussed.

This Chapter introduces the idea of ethnomethodology and positions the talk-in-interaction of software developers as a matter which merits detailed

study. Ethnomethodology provides an analytic framework within which social activities, including those at work, can be studied. This Chapter examines why an ethnomethodological approach is useful when studying work and how it might be applied to the activities of software developers. It should be noted that whilst some ethnomethodologists choose to ignore the broader context within which their informants are working, preferring to focus on the ways in which the immediate context is made relevant by participants, this is not the approach which will be taken here. The activities of professional developers are of interest precisely because they are situated within organisational contexts, because they are oriented to the accomplishment of work-related goals and because they relate to the expertise of individual workers.

This Chapter will discuss a range of literatures from ethnomethodology, conversational analysis and studies of group working to reveal relevant ideas and arguments from them. The operationalisation of these ideas is discussed in Chapter 4. This Chapter starts with an examination of ethnography and its use in studying work, 3.2. Section 3.3 introduces the ethnomethodological perspective on work. Section 3.5 discusses how meaning is formulated in the communications of developers and Section 3.7.2 considers how shared understandings might form within teams as part of their common culture.

3.2 Ethnography

Ethnography is perhaps the most commonly used of the subjective research methods within software engineering. This Section introduces ethnography, particularly as a way of understanding work, to show that it provides an approach to understanding the work of professional software developers which

is rich in detail and analysis.

The researcher who is in the field studying work is immersed in an often alien culture, observing and interrogating members of the culture to uncover the ways in which they produce, understand and sustain that culture. This approach to research is, broadly, *ethnography* but, more specifically ethnography is a "written account of a culture (or selected aspect of a culture)", (Van Maanen; 2011).

The subject of an ethnography is *culture* and its method, the way in which data are gathered, is *fieldwork*. The essential purpose of ethnographic writing is to pull together the fieldwork and culture in such a way as to reveal details of the culture to outsiders, or sometimes to its members. In so doing, ethnographic accounts may reveal the choices and restrictions which are the heart of social lives, (Van Maanen; 2011). Anderson (1997) writes that many social scientists blur the boundary between fieldwork and ethnography but that the latter is an "analytic strategy for assembling and interpreting the results of fieldwork".

The idea of "culture" may not seem to be immediately applicable to the workplace but workplaces, companies and even individual teams each have their own cultures. Developing an understanding the work which they do, how they do it and why they do it means developing an understanding of their unique workplace culture. Ó'Riain (2008) demonstrates that workplaces have these unique cultures when he writes, of a project he is investigating, that "[t]he team takes on a culture of its own, manifested in the mimed hostility to [managers] suggestions but also in the information-sharing, problem-solving and solidarity building within the team on an everyday basis". Soft-

ware developers have used ethnography to study the cultures of organisations for many years as part of their requirements gathering or usability processes, (Suchman; 1987, Dourish and Button; 1998). More recently academic software engineers have begun to use the same approach to understand aspects of the software development process, (Sharp and Robinson; 2003, Lethbridge et al.; 2005).

In this research, the researcher is an experienced software engineer. Entering the workplace means entering its culture and observing developers at work means observing them as they orient to that culture. A researcher who knew nothing about software engineering would observe and be interested in a different set of phenomena to those which a fellow developer would see. Some aspects of work which are mundane or predictable to the insider might be of intense interest to an observer who was not familiar with the discipline. For example, the use of test-driven development says something about the ways in which developers organises their work to another engineer. To an outsider, the whole idea of testing could be something which is worthy of deeper interrogation.

The outsider is unlikely to have sufficient time to be trained in programming or testing and hence can never be a participant who is fully immersed in the workplace culture, (Crang and Cook; 2007). In these studies the fieldwork was done by an insider who naturally became a participant in the work through conversations with the developers about their work. This insider perspective is a different one to the perspective of the stereotypical anthropologist who spends years observing, understanding and gradually joining a culture.

Van Maanen (2011) writes of fieldwork as the quintessential ethnographic

activity and that analysis only has credibility if it based on things which the researcher has seen or been told by a member of the society. Data from the field give ethnographic writing its credibility and, Anderson (1997) argues, legitimise the ethnographer by grounding their analysis in empirical data. Those data are gathered in myriad ways. In the studies which are presented here, the data sources are contemporaneous field notes and audio recordings with the former being the primary source for ethnographic writing.

The attraction of going into the field to see what users do, or to see how developers organise their work, is that it gives rich data which are embedded in context and which can be written-up in ways that are both interesting and insightful. Such writing could become a journalistic re-telling of scenes which were observed and conversations which were recorded but ethnography is not this type of straightforward write-up of field notes. When ethnographers leave the field they do so to "write up the culture" (Crang and Cook; 2007). Such a write-up presents a detailed understanding of that culture including an understanding that "things are not what they seem" and that appearances are often deceptive, (Anderson; 1997).

If appearances are deceptive, members are unlikely to have a comprehensive understanding of their own culture which is why the ethnographer does not simply ask them for their explanations as a journalist might. Instead members' accounts are interrogated alongside the researcher's observations, sometimes recorded in contemporaneous notes, other data sources such as audio or video recordings and sociological or anthropological theories of culture. The ethnographer tries to produce a rich picture of a culture, a picture which is supported, and which supports, academic theories and positions.

The write-up is not a value-free neutral activity, it requires that the author undertake a *post hoc*, analysis and interpretation of the data, (Anderson; 1997). Researchers such as Sharp and Robinson, who are immersed in the domain which they study, have their own cultural values which, necessarily and rightly, inform their analyses of their data. The impact of this on software engineering, especially when using ethnography as part of requirements gathering, is the death of certainty about "facts".

It is important to recognise that debates about status, legitimacy and meaning exist. A post-modern deconstruction views ethnographic writing as subjective literature rather than as objective science, (Linstead; 1993). For post-modern ethnographers their texts are, Linstead argues, active descriptions rather than neutral recordings of the worlds of "others". A single authorial voice is replaced with a multitude of voices in which all interpretations are possible and rigour is knowingly and willingly, lost.

Whether an ethnographic account is read as a neutral account of a culture or as a living representation of that culture, it shows something of the culture. Ethnography is, ultimately, both a rich description and revealing analysis. When working life is studied through ethnography the workers and their practices are the principle matters of interest. This differs from traditional studies of work which can be exercises in organisational structures and power relationships as they are codified in "org charts" and human-resources documentation.

When observers look at work they can easily lose sight of the worker and of the work which they do. By using ethnography and, specifically, by looking at the sense-making activities which form the basis of the work its "hidden"

nature can be shown. Two ideas inform such a study: Goffman's idea that people's talk is a social domain which can be studied as an institution in its own right; and Garfinkel's idea that the production of sense in talk is due to "ethnomethods", (Heritage; 2008).

The following Sections introduce the ideas of ethnomethods, sense-making and conversation analysis.

3.3 Ethnomethodology

Scholars of work often focus their studies on large problems of structure, of management or of process but if work is to be understood then the actions of the workers must be examined in detail, (Llewellyn and Hindmarsh; 2010). Ethnomethodology originates in work which explicitly confronts the study of the structures and functions of societies by "respecifying" the production and accountability of those societies, (Garfinkel; 1996). Garfinkel's respecification was that social order arises from, and is made to work by, the actions and interactions of its members, (Dourish and Button; 1998), and that the production of this order is *accountable*. Ethnomethodological accountability means that as a basic grounding of everyday activity we each strive to understand the actions of others and to make ourselves understandable and explainable to them, (Suchman et al.; 2002).

Ethnomethodology, following Garfinkel, takes as its analytical locus the actions of the members of a community which are analysed to understand how those actions create stable social orders. For ethnomethodologists "there is order in the most ordinary activities of everyday life in their full concreteness, and that means in their ongoing procedurally enacted coherence of sub-

stantive, ordered phenomenal details", (Garfinkel; 1996). By examining the raw details of the structure of typical social phenomena the social order can be revealed.

The production of social order is "part of ordinary, everyday life, woven into the fabric of all activity", (Dourish and Button; 1998). Developers gathered in a daily stand-up have a mutual interest in constructing the meeting as a stand-up and must orient to it as such. Their actions, their talk, should, reasonably and usually, be expected to be about their work since that is the matter of interest around which they have gathered, (Dourish and Button; 1998).

Ethnomethodology is interested in *sense-making*, the ways in which mutual understanding is achieved. In its methods, ethnomethodology examines those taken-for-granted meanings and assumptions which underpin members' social actions. Garfinkel's work revealed "that meaning requires order, and the empirical elaboration of how this is achieved through sequential devices and reflexive attention, are [his] unique contribution to social theory", Rawls (2008). If social order is required for there to be meaning, then the context of the production of the action is important because the order is created within that context. Actions are located within organisational structures whose boundaries, both internal and external, affect the ability, or willingness, of members to understand each other, (Suchman; 2003, Orr; 2006), and so define the context of accountability. Boundaries, whether explicitly defined or implicit, such as the ones described between managers and technicians in (Orr; 1996), impact upon mutual understanding but do so through their impact on members' accounts, (Rawls; 2008).

Although a manager and a technician or a programmer and a sales agent

may not understand each others' technical jargon, they are able to interact successfully because of their ability with conversational techniques. Garfinkel was "concerned with the patterned and instructable ways in which order properties of situated action are made public and mutually recognizable objects by workers at worksites from the contingencies at hand", (Rawls; 2008). People engaged in conversation have an interest in preserving mutual intelligibility. The methods used to ensure understanding, and their use, becomes the object of analysis. Whilst much talk is routine in structure or content, in any particular work situation the talk will be situated and contingent upon the working activity at that moment. This research shows software developers talking as a sense-making activity – they talk to each other to help them understand problems, pieces of code or the design of software. Their talk is not only situated within their work activities, it produces those activities.

Ethnomethodology provides a theoretical orientation to interaction which reveals actions and their sequencing and shows how workers orient towards these as they do their work. At the same time the ethnomethodological analysis is located within the context of the production of the actions and preserves the contingencies of that context. A context is not a fixed set of social or cultural factors which act from the outside upon a situation, rather it is a set of reflexively produced relationships between actions and the meaning of those actions in a specific place and time, (Lynch and Peyrot; 1992). For example a semi-formal stand-up meeting as part of a Scrum or an *ad-hoc* conversation between two programmers provide a situated context for their actions. Events such as stand-up meetings can be understood as concrete phenomena, ethnomethodology argues, because the people involved are constantly reproduc-

ing them in that way, (Llewellyn and Hindmarsh; 2010).

What Durkheim and Coser (1997) call the *shared competencies* of developers' in programming and its associated activities provide the *lingua franca* of software development. They become a basis from which ideas can be both sustained and challenged. Many software developers work in distributed multinational, multi-cultural teams in which the culture of software engineering and its situated practices provide the only common framework on which the social order of the team or project can be established, (Grinter et al.; 1999, Ye et al.; 2004, Kotlarsky and Oshri; 2005).

The idea of *indexicality* is central to ethnomethodology. Indexicality is the property of some expressions to mean different things in different situations, (Suchman; 1987, Dourish and Button; 1998). Expressions such as *the user* or *the database* change their meaning within conversations, (Rawls; 2008). Many reasonably complex pieces of software connect to numerous databases but the documentation may not refer to *the user database* or *the customer details database.* Instead the particular database and its use must be inferred by the reader from the context within which the phrase is used. Context is complex and important because almost anything we say can be interpreted in different ways depending upon the situation. However the context of each statement which we make is not included in the statement itself, instead we "wave our hands" at it, (Suchman; 1987).

The ethnomethodological meaning of indexicality goes beyond the context within which statements are used. In his early writings Garfinkel stressed that indexicality requires *action* as speaker and listener work together to create context, (Atkinson; 1988, Goodwin; 2000). By studying talk as it is produced

within the context of its production the ethnomethodologist is able to reveal the reasoning and understanding which it embodies, (Gephart; 1993). But indexical expressions are "specifically ordinary and uninteresting", (Garfinkel; 1996). Whilst they can readily be observed, Garfinkel stresses that these expressions should not be subjected to a cognitive analysis. The analyst can see that the expression is actively working for both the speaker and the hearer but cannot make the leap from there to understanding *why* that work is being done.

If the concept of indexicality is extracted from the study of speech it can be used to reveal the understanding which is developed and shared in the workplace. Revealing shared meaning is an important part of many workplace studies whether Latour (1987) studying how science is created, Heath and Luff (2000) looking at the use of control systems, Orr (1996) following repair technicians or, radically perhaps, Faulkner and Brecker (2009) describing the making of jazz music on the concert stage. Each of these studies demonstrates work in which communication is central to the activity as it is lived by its participants. In each case before there can be communication there has to be understanding. The ways in which people interact during their work builds a shared context within which they are working.

3.3.1 Conversation analysis

Conversation analysis was created by Harvey Sacks in collaboration with Emanuel Schegloff and Gail Jefferson in the 1960s under the tutelage of the Harold Garfinkel, (Heritage; 2008). Sacks was interested in *practical reasoning* in institutional settings such as police work or psychiatric counselling. Using tape

recordings, transcribed and annotated using a scheme developed by Jefferson, of phone calls or meetings Sacks and Jefferson were able to reveal the practical work within the talk-in-interaction.

Sacks, Schegloff and others moved from studying talk in institutional settings to studying general features of talk and interaction such as turn-taking. Other conversation analysts were interested in studying talk within institutions to discover how those institutions were produced, (Heritage; 2005).

Conversation analysts studying institutional talk link the meaning of talk to the context of its production. The meaning of an action is formed within the context of previous actions and the social context for interaction is created dynamically through the sequential ordering of interactions, (Heritage; 2005). This then leads to a theory of the ways in which participants orient to an interaction:

- Participants construct talk within the context of previous statements.
- When people construct an utterance they have an expectation that the response will come from a limited set of possibilities. They are creating the context for the response with their own statement.
- Responding to an action, a statement, requires and demonstrates an understanding of previous actions.

Conversation analysis has been a productive methodological application of ethnomethodology. The two have different agendas: the former being interested in the sequential nature of talk, (Heritage; 2008), whilst the latter is interested in "mundane reasoning", (Atkinson; 1988). The ethnomethodological project seeks to understand work as a mundane, observable activity in which the sequence of actions, activities and talk reveal both the work and

the participants understanding of it as it is embedded in social interaction. This concern with detail can, Atkinson argues, mean that ethnomethodological analyses can "merely recapitulate the observed sequences of activities with little or no framework for selection, or for the representation of those activities in any other discourse".

It doesn't have to be the case that analyses of conversation are analyses of language, structure and sequence. Hymes (1994) writes that "it is not linguistics, but ethnography, not language, but communication which must provide the frame of reference within which the place of language in culture and society is to be assessed". Much good work in conversation analysis, goes far beyond description, providing illumination of institutions or situations which would otherwise be only through common sense understandings. Studying doctor-patient relationships, counselling, classrooms and so on has "drawn attention to the detail and complexity of everyday life, and to the delicacy with which participants monitor the unfolding conversation as they collaborate in its production", (Atkinson; 1988). The context is important because "the boundaries of the situations within which communication occurs... are conditioned by properties of the linguistic codes within the group, but are not controlled by them", (Hymes; 1994).

In conversation analysis, the ethnomethodological interest in indexicality becomes a concern with conversational structure and sequence. Ethnomethodological reflexivity becomes an interest in the work which is done by individual statements and sequences of interactions, (Potter; 1996). The structure of conversational interactions are neither accidents nor artefacts but are crafted by speakers with a sensitivity to both the sequential context of their

production *and* to their role in the interaction. Within conversation analysis the "messy" details of delivery such as changes in intonation, pauses, repetitions and so on become matters of interest in a way that is fundamentally different to the techniques used by linguists. These details matter because they are used by speakers as part of their method for interacting and hence they are there to serve the action which the speaker is performing, (Heritage; 2008).

Analysing the things which people say to each other about their work can reveal their understanding of that work. Sharp et al. (2004) write that "[i]t is via language-in-use that people reveal, perhaps inadvertently, implicit knowledge and meaning. This knowledge and meaning might well be different to what their companies would like to portray or what they themselves might rationalize". Conversation analysis reveals more than surface facts, it exposes ideas, opinions and tacit judgements which might be rationalised away or ignored in a higher-level analysis. However, that analysis should be grounded in "categories" which might be relevant to the participants, (Hutchby; 1999).

Annotated transcriptions of conversations between developers are given in Chapters 7 and 6. Details of the annotation scheme are given in Jefferson (2002) and in Appendix A. The scheme used here is taken from Ten Have (2007) and is a subset of Jefferson's complete set of symbols. This subset was chosen because it is rich enough to provide coverage of the features of talk which are required for these analyses yet is sufficiently simple that the transcriptions remain clear to non-specialists.

3.3.2 Face-work

When people interact they perform both verbal and non-verbal acts through which they express their view of the situation and of the other participants, (Goffman; 1964). Together these acts form, for each individual, a *line* through the interaction which a person uses to claim social value. Goffman calls this claimed social value *face* and identifies it as something which might wish to maintain, if they "feel good" about a situation or change if they "feel hurt". Participants are not only interested in their own face, they have feelings about and for the face of those with whom they are interacting.

Face is constructed in the moment but is built on a history of interactions. Goffman (1964) writes that a person maintaining face in one situation is "someone who abstained from certain actions in the past that would have been difficult to face up to later". The current interaction may be dependent on wider social considerations if those involved interact repeatedly but if they will not meet again they do not need to worry about maintaining face for each other.

The case studies in this research are focused on teams of colleagues who work together every day. It is important that they are able to maintain their own face and that they do not damage the face of their colleagues. Goffman writes that when people are "in wrong face or out of face" they feel inferior to those around them, especially if they were relying on the encounter to "support an image of self" which is now threatened. Bargiela-Chiappini (2003) expands this idea by noting the role which emotion plays in interactions "so that harm to another's face causes 'anguish', and harm to one's own face is expressed in 'anger' ".

Because of the consideration given to the face of others, face-work is re-

lated to politeness. But face-work is about "self-presentation in social encounters", (Bargiela-Chiappini; 2003), and is a set of actions which make the individual's actions consistent with their face. Politeness is about "rational, goal-oriented behaviour", "politic behaviour" or "appropriate behaviour" according to Bargiela-Chiappini (2003). Face is found in the interactional order, (Goffman; 1964), whilst politeness derives from the social rules governing interactions, (Bargiela-Chiappini; 2003).

The importance of both face-work and politeness in Daily Scrum Meetings is shown in Chapter 6, and face-work when pair programming is shown in Chapter 7.

3.4 Negotiating design

Programming is a collaborative, social activity which is not only performed at the computer. Programmers work in teams, they talk to each other, they communicate more widely with customers or users and, wider yet, with communities of programmers at conferences and on Web forums. Aspects of this have been studied, often as part of the design process in the creation of programming tools or in computer-supported cooperative working projects. Software development has many features in common with both engineering and design, both of which have been extensively studied.

Studies of collaborative design, engineering and software development have often taken a task-oriented approach in which conversational practices, representations such as sketches and the use of shared references are worthy of study only as they orient towards the completion of the task-at-hand, (Cahour and Pemberton; 2001). When design is studied as a social practice both the

aims of the researchers and their analytic approaches have more in common with Conversation Analysis of non-technical situations. In studies of, and by, designers such as Cross and Cross (1995), Cross (1997), Cahour and Pemberton (2001) "the goals of participants, whether explicit or not, will be not only the creation of a design representation but also the management of interpersonal relationships". These studies show that transactional talk and interpersonal talk are woven together throughout conversations in a complex relationship. Interpersonal talk can inhibit task-oriented talk if the need to manage the social situation is strong so that, for example, a developer may accept a weaker solution to save another participant's face, (Cahour and Pemberton; 2001).

The problems on which product designers work are, as with those which concern software developers, often ill-defined so that "analysing and understanding the problem is an influential part" of the process, (Cross and Cross; 1995). The development of understanding is widely recognised to happen through a cycle of talk-based *propose-evaluate* iterations, (Cahour and Pemberton; 2001), which are, at least superficially, similar to the cycles of an agile method. Cyclical design processes inevitably means that the design remains changeable and changing well into the process. Ronkko et al. (2002) shows that in software development even the naming of variables is contingent, temporary and subject to change. There is a cost associated with the constant changes which teamwork can bring but teamwork is likely to lead to a better overall solution. A team will propose more potential solutions than can be generated by an individual. Not only is the solution space larger, the team must work together to negotiate it as they move towards a final design. Collaborators must identify, avoid and resolve conflict as they search for a solution, (Cross and

Cross; 1995).

Incomplete specifications are common in engineering. An engineering design manager interviewed in Lloyd (2000) describes one process for handling the problem. The requirements documents which the sales team produces are so vague that a process has to be invented to handle them: "for every new project that comes in I'm going to put a requirement for a variance document. The reason I'm doing that is because the orders come in so open—the one pitch quote". But the sales documents are so vague that the engineers have to "you know, get on the telephone, or go and talk to the people involved to find out what in fact is really required, to get some idea of, you know, to try and summarise it in some kind of simple way". This is an *ad hoc* process developed on the ground to handle a specific type of problem in the company. Because the process is informal it isn't documented and could easily be glossed over by the manager and engineers who use it. Conversation analysis gives us a way to understand how the engineers create usable requirements and make decisions about products.

Problems of misunderstanding, poor communication or lack of trust can be seen across organisations at all levels. For example, Laine and Vaara (2007) examine the understanding of strategy in an engineering consultancy. They show that there is a difference between the strategic discourses of the management and the interpretations of, and orientation towards, those discourses by consulting engineers. The various discourses which develop around organisational strategy are important because different discourses are used to create and support positions around strategy. Much research into strategy takes a top-down managerial view whilst silencing or, at best, side-lining al-

ternative voices. Laine and Vaara reveal those voices and in so doing reveals an organisational "battle over power, hegemony and individualized sense of identity". In studying software development we should expect to see multiple discourses around sales, functionality or approaches to engineering. Such discourses can be revealed by studying the ways in which they are embodied in, and accountable to, the working practices, documents and, most clearly, the talk of members of the organisation.

In a software project with specialist requirements analysts liaising with the client, those analysts are assumed to understand what the developers know without having to ask, acting as buffer, translator and clarifier. The "members of the requirements development groups were experts in their fields, in some ways more expert than the customers. Developers used this expertise to make sense of the incomplete and ambiguous input they received from customers and translated it to their own domains", (Crowston and Kammerer; 1998).

The informants in Crowston and Kammerer (1998) represent an ideal form of analyst for whom misunderstandings are minimised and for whom the worlds of developer and end-user are equally transparent. Usually understandings are contingent upon external factors such as the prior knowledge, skills and experience of the analyst or developer. An experienced developer will use a different approach to those which are chosen by developers with less experience, (Détienne; 1995), and that approach may not depend upon the problem which is being solved. The generic structure of the solution may be the same regardless of the particular problem. This might seem surprising but apparently different applications often have similar architectures and the documentation produced by analysts is frequently out of date by the time that

code is written, (Lethbridge et al.; 2003).

3.5 Finding meaning

Project teams have to share large volumes of information, and the larger the team, the greater the volume of information which must be shared. Communication can be especially difficult when products are built by distributed teams. Although team members can talk to each other using a variety of media, ranging from chat systems through email and video conferencing, communicating meaning can be difficult. Teams which work as cohesive units become, in effect, isolated communities, developing their own ways of talking about their work. Over time these different approaches to talk can become different languages which are mutually incomprehensible. Such teams form *communities of practice*, an idea which is discussed in Section 3.7.5.

When members of the team talk to members of other teams they have to reach a shared understanding of the conversation. If they fail to do so each will leave with a different interpretation of the conversation, its meaning and its outcome. This is not a problem of technical language: the vocabulary of software development is understood across roles and skill sets and becomes the *lingua franca* of parts of the project. Everyone on the project will know what is meant by *class* or *entity*. Much of the talk during a project uses language which is less precise and is more open to interpretation because the architecture of a piece of software is a plastic construct, (Smolander; 2002). Architectural decisions, even the very concept of a software architecture, have different meaning for different stakeholders. These meanings are both concurrent and divergent which makes them a source of misunderstandings and

mistakes.

It is not only new code which requires that the programmer understand what was expected and why. Much of the effort across the life of a piece of software is maintenance which can be anything from fixing bugs to meeting changed requirements and on to adding whole new areas of functionality. Maintaining code requires detailed understanding of it both as an artefact and as a historical document which expresses the purpose behind its original creation and the set of changes which have so far been made to it. Seibel (2009) quotes Simon Peyton Jones of Microsoft saying "[o]ne of the most depressing things about life as a programmer, I think, is if you're faced with a chunk of code that either someone else wrote or, worse still, you wrote yourself but you no longer dare to modify". Software engineering has made little progress since 1969 in helping programmers maintain code, (Glass; 2006a). Each programmer has to use the code, any documentation they can find and the help of any willing colleagues they can find as they search for meaning. Useful changes can be made only once the programmer understands the source with which they are working.

Many programmers consider that the source code is the canonical truth of the system on which they are working. Code, necessarily, explains its own inner workings but even well commented code does not explain how or why it was written. The history of the code is often invisible so that "knowledge about the code and the design decisions remain in the head of developers", (Nakakoji et al.; 2006). Full understanding of code, even when documented, can only come through revealing the context of its production: why it was made, how it was made and where it failed to meet its original specification,

(Banker et al.; 1998).

The context within which code is produced includes knowing the context and culture of the group producing it sufficiently well to reveal hidden meanings in the idiomatic forms they use. To develop or maintain code based solely on a requirements document, the developers needed to acquire membership of the group which produced that documentation. Without such membership they are constantly searching for understanding. Documentation is, in ethnomethodological terms, *indexical*: it only makes sense within the context of its production. If the program is to be a solution to the problems which the requirements documents present the developers must have *adequate indexicality*, (Ronkko; 2007).

Software is almost always subject to some form of design representation before it is implemented as code. These representations, often written works, range from long text documents which describe the functioning of the finished systems and the constraints upon it, shorter written *user stories*, which describe how a user will interact with specific parts of the system, though to jottings on post-it notes. Alongside these writings software engineers use a myriad of diagramming techniques to create visual representations of the system's architecture (UML) or its graphical interface (wireframes). In recent years there has been a move to creating prototypes, partial implementations, to demonstrate understanding. "[L]iterary devices continue to have their uses, but the centre of gravity shifts from the production of system specifications and various other abstract renderings of system functionality, to the prototype and associated practices", (Suchman et al.; 2002).

Any representation reveals different meanings at different readings. A

wireframe diagram is adequate when talking to a user but may be less so when used as the basis of a graphical interface, a long text document may be prefect for an audit of a project but lack detail when given to a programmer. Prototypes are regarded as useful because they can be used to uncover work requirements, technological possibilities and the (mis)understandings of both users and developers. Thus a prototype becomes a device for the creation of better indexicality than that which can be created through the use of either diagrams or text documents. Cost and time constraints mitigate against the use of prototyping on many software projects.

3.6 The role of representation in talk about programming

Understanding of software projects is built on more than talk. Projects generate extensive documentation ranging from formal statements of requirements through structured diagrams and even post-it notes on an informal Kanban board. All of this documentation is used to share information about the project and its product and to help the developers co-ordinate their work.

During development the most useful documents may be diagrams. As students, all programmers learn to create diagrams of both problems and solutions. Drawing the system becomes a natural part of their search for the adequate indexicality which Ronkko (2007) identifies as necessary. Drawings, whether diagrams, formal UML models or *ad hoc* scribbles, establish relationships between the parts of a program, (Suchman and Trigg; 1996, Blackwell et al.; 2001, Bates et al.; 2011). Dittrich and Rönkkö (2002) followed a team of student developers as they work on a large project and found that "they achieve a physical sharing of key objects with the help of drawings on the

whiteboard".

What does it mean to look at a set of diagrams and at the code which was created from them? How does the programmer come to understand what the structure of the code means – and how they might alter that structure without breaking it? These are questions which form part of developers' everyday talk about their work. And they are answered by thinking through the various representations that the developers create.

Suchman and Trigg (1996) follow two researchers as they design an artificial intelligence application. They show how ideas are shared and developed both through the creation of transient informal drawings on a whiteboard and through the developers' talk around and about those diagrams as they are produced. They call this process "socially organised craftsmanship", reformulating technical activity as craft almost a decade before the rise of the software craftsmanship movement. Formulations such as this have been used elsewhere to characterise other technical activities as craft. The laboratory work of scientists may appear to be structured and controlled but when examined closely it has many similarities to common-sense notions of craft, (Latour; 1986, Sennett; 2008).

Scientists, programmers and craft workers use informal representations as physical manifestations of their thought process. Blackwell et al. (2001) call these ephemeral representations *talking sketches*, diagrams which provide a focus for discussion between colleagues.

The process of creating and manipulating physical objects such as drawings gives those thoughts and processes what Latour calls "immutable mobility" and renders them persuasive because of the "reflexive relation" between

their production and use. Documents such as drawings which are created or used as part of a development have meaning beyond the moment of their creation. They encapsulate a process at a specific time and place. The document will be moved and used later, possibly by many different people but carries with it something of when and why it was made.

Suchmann and Trigg write that "devices for seeing" are particularly relevant to studies of science. Documents such as drawings or notes "stand for the structure of an investigated phenomenon". For programmers the tests which they write, the backlog entries they maintain, their code and even talk-about-code are devices for seeing. When developers document their code or talk to colleagues about it, the explanation which they give stands in lieu of both the code and the intent of the program.

When Suchman and Trigg's informants create artificial intelligence software they are encoding their own lived experiences and common-sense understanding of the world in symbolic systems which "delegate human competence to machines". This encoding requires that the problem be simplified and that the process of simplification happens through negotiation. AI research is characterised in this paper as being a two-stage process: in the first stage the researchers take activities which are "thorny problems of representation" and try to understand them; in the second stage they encode those activities symbolically in computer simulations. The same is true for most programmers. Complex problems are represented symbolically, for example in a UML diagram, before being converted into a different symbolic form as code before being reified as a functioning program within the computer.

The scientists use whiteboards as their primary "representational technol-

ogy" and such boards are ubiquitous within their lab for this purpose. Having a common writing area helps a team, be it a wall, (Evans; 2003), a table top, (Weinberg; 1998), or post-it notes and scraps of paper. In a sense the medium doesn't matter since it provides a locus for the real work which is being done in the developers' talk-about-code. The drawings on the whiteboards are largely meaningless to an outsider but, ethnomethodologically, their production is interesting and meaningful. The symbols and notes on the board may not make sense to a casual passer-by after the discussion but when seen "in relation to the activity of their production and use... they come alive as the material production of 'thinking with eyes and hands' that constitutes science as craftwork", (Suchman and Trigg; 1996).

Representation is at the core of all stages of software development. One of the main themes of academic software engineering has been the creation of suitable representations. These range from mathematical notations (such as Z) to diagramming techniques (UML) and on to programming languages. Software engineers understand that the representation affects what is represented. Where the cost of using a representation is great, as with maths, it will be used sparingly. Representations which are lightweight yet expressive will probably be used more often and more effectively as is seen with post-it notes on a Kanban board. Suchman and Trigg develop an anthropological sense of such representations in which they are part of "a socially organised activity producing certain publicly available artefacts".

The question of *what* to represent is a matter of both the problem domain and of the expertise of the developers. In Suchman and Trigg the domain is of a research program, for the developers in this study the domains are a vari-

ety of embedded and Web applications. The types of representation made in each case will differ because the cases themselves differ: the plasticity of software results "in relatively greater freedom regarding the form of the diagram elements", (Blackwell et al.; 2001).

Talk arises out of and around representations. Software is composed from myriad complex systems as well as the code which the developers create. That is, the final product builds on code in libraries and uses other systems through API calls and both process and network messages. Part of the talk of developers has to be to manage and control this complexity. Experienced developers perform this control by applying prior knowledge but the team needs to talk this through with types of talk which describe, explain, argue for solutions or articulate the work, (Dittrich and Rönkkö; 2002).

3.7 Communication and coordination

Software projects are complex ecosystems built on relationships between the artefacts being produced, the individual developer and a wider developer community, (Nakakoji et al.; 2006). The importance of the artefact as the locus of meaning was discussed in Section 3.5, the importance of the individual is a matter of education, training and psychology which lies outside the scope of this work. In this Section the importance of the development community, in particular of the team, is examined.

The structure and functionality of the final software arises from negotiation within a group of customers, users, management and developers. This group collectively arrives at a vision for the application as they work together. In creating code the developers build upon a wide range of resources includ-

ing their own fundamental knowledge, complex development frameworks and trends or fashions within the industry.

The systems which teams build are solutions to customers' problems but these solutions have to be created within pre-determined budgets, timescales and technical constraints. These external determinants constrain any negotiation around the structure or functionality of the software. The placement of those constraints and the effect which they have on the shape of both the project and the product arises from negotiation. Those constraints are subject to a range of discourses which act upon the developers' own ideas about how the system might be implemented to create a solution space and, ultimately, a final solution for the problem-at-hand.

Software is developed by teams whose members have diverse skills and knowledge and who may fulfil more than one role within the team or who may work across a number of teams. Modern software development teams are often distributed across sites, sometimes even across countries or time zones. Distributed teams are not necessarily problematic for their members or their managers. When the team members orient positively to distribution and the use of tools such as video conferencing they can work effectively. More typically, though, distribution, especially across time zones brings a range of problems of understanding, coordination, trust and ownership, (Sillito and Wynn; 2006).

Coordinating the work of a team is a social problem rather than a technical one, although one which is often given a technical gloss through the use of tools such as backlogs, IM or video conferencing. Sharing information doesn't always improve communication. One development manager quoted by Sillito

and Wynn (2006) said that in their organisation "everyone is inundated with email and newsletters. We just scan them. So how do you get the word out there?". The code which many teams build has to be used by other teams, either immediately or in the future. Both requirements and the resulting code need to be consistent and to meet the needs of all teams. This becomes a fundamental problem of coordination, of understanding and of effort.

Coordination can be supported by tools which let developers find the information they require when they require it. Backlogs, Kanban boards, in-code comments and formal documentation are all ways of sharing information in which the information is *pulled* rather than *pushed*.

3.7.1 Organising teamwork

Team work is difficult. Simply putting developers into a "team" will not make them more productive – most teams deliver software which is substantially late or over budget, (Teasley et al.; 2000). Improvements to the performance of a team require that actions be taken which impact upon the workings of the team. Examples include "radical co-location", (Teasley et al.; 2000), or the original XP project, (Beck; 2000), which both had the whole team working together in the same room, or radical exposure to Scrum including a daily Scrum of Scrums, (Sutherland et al.; 2006). These approaches, and many other, have shown that massive gains in productivity and reductions in costs can be associated with entirely unconventional ways of working.

The literature around radical ways of organising software development, and here Agile approaches are not necessarily radical, tells us that coordination and communication are the areas of working practice which can be

changed most radically and from which the most benefit arises. Coordination moves from a function of project management or team leads to become a function of everyone on the team. Developers share information about their work and, as a result, everyone on the team is able to find out who is doing what, when they are doing it and how it will impact on their own work. One of the major drivers behind information sharing is a reduction in the time between taking a decision and implementing it, (Cockburn and Highsmith; 2001).

When people talk about their work as part of that work, they typically have a specific goal in mind. They are gathering or sharing information or coordinating tasks with, or for, colleagues. Such *task oriented* talk differs from talk which is used within work but in more social ways, (Holmes; 2005). Task oriented talk is more focused and has different structures which are related to the nature and complexity of the task. More complex tasks engender more complex patterns of communication, (Tushman; 1978, Tschan; 1995).

Like most knowledge workers, software developers talk in different ways to achieve different goals. Information seeking is a different practice to asking for help which is different, in turn, to relaying generally useful information. A developer seeking information may ask "is anyone using the build server?", if she asks for help she may want to know "how do I abort a build?" whilst in giving information she may offer an unprompted "the build server is free now". Each statement may be used to initiate a conversation or as part of an ongoing conversation. Each is potentially problematic in its own way – there is a risk to one's face in asking for help or for information, there is a risk to other's face in offering unsolicited information – but it is a problem space which has to be negotiated many times during each working day.

Traditional engineering disciplines have a culture which mitigates against information seeking and sharing. Engineering is seen as "individualistic" and "macho", (Eonardi; 2003). Software engineering appears to be different in that the culture of the discipline is a culture of sharing information, (Kotlarsky and Oshri; 2005, Ó'Riain; 2008). However this sharing is not simply a matter of talking to colleagues.

In seeking information the relational aspects of the interaction are as import as the transactional ones. Generally developers prefer to use informal information sources such as conversations with colleagues over formal sources such as documentation, (Milewski; 2007). The cliched developer who is more interested in technical aspect of their work than in the social ones does exist. Sometimes those people are the very experts who need to be consulted most often but they can easily be sidelined, if their communication skills are poor, (Skowronski; 2004).

3.7.2 Communicating within development teams

Once a developer decides to make a change they need to be able to communicate it to their colleagues, assess its impact and evaluate its cost. Communicating change, and planning its impact, requires effort and the application of a suitable strategy. The developer making a change needs to be aware of the impact of that change on their own work and on that of their colleagues and clients. One important variable is the "size of the impact network, the set of software developers being impacted or impacting a specific developer", (de Souza and Redmiles; 2008). A large change may have to be communicated widely and may need to be rolled out in a coordinated way so as to minimise

its impact on the work of others throughout the project.

Individuals can only coordinate how they work if they are able to share information freely, cheaply and quickly. "[I]nformal, unplanned, ad hoc communication is extremely important in supporting collaboration", (Grinter et al.; 1999). The costs of coordination here are measured in the effort which is required to enable it. Those costs can include "cultural and language differences, trust and commitment, extended feedback loops, asynchronous communication, and knowledge management", (Layman et al.; 2006). However, easy, informal access to colleagues provides valuable benefits in both coordination and information sharing, (Kraut and Streeter; 1995, Cherry and N.; 2004). Oral communication is the least formal form that can be used. It tends to provide information and context which are more current and more relevant than those provided by any other medium, (Tushman; 1978).

If communication is made easier then coordination can improve and the consequent benefits in quality and productivity will follow, (Gopal et al.; 2002, McChesney and Gallagher; 2004). One thing which can improve communication is to have people who need to talk to each other working near each other. In organisations which introduce agile methods, managers often require co-location of the development team, (Boehm and Turner; 2005). The original XP team worked together in a single room to facilitate communication, (Beck; 2000). When teams are co-located, changing the structure of their office space to facilitate communication can improve their performance. The use of white-boards, placing desks next to each other and removing partitions can all help to simplify communication, (Sharp and Robinson; 2004).

Even if the team doesn't work together all day, having a space in which they

meet facilitates their work. Sawyer et al. (1997) examined a company in which teams were given access to a computer-supported team room. Teams would use the room to examine and discuss designs and code. Use of the room meant that group discussions became easier, "[t]he direct effect is to make it easier for developers to work together; enabling the production aspects. So, software development improvements at this site have emerged without increased engineering. Rather, they have emerged due to increased discussion".

Distance is a major disincentive to communication. When projects are geographically dispersed communication between parts of the project are often difficult. Herbsleb and Mockus (2003) give many reasons for this: staff at different sites are less likely to identify as being on the same project; it is difficult to identify remote colleagues as expert; the view of priorities will differ between sites. Staff at each site can engage in *group think*, (Moorhead et al.; 1998), and develop ideas about the project which are different to those held elsewhere. The tendency to think in a similar way to those with whom one is surrounded can, Moorhead et al. posits, lead to defective decision making by the team.

Even with increasingly ubiquitous technologies such as instant messaging, on-line forums and video conferencing, people are far more likely to communicate with those who are local than with those who are distant. Once the distance between two people passes just thirty metres they are no more likely to communicate than if they were on opposite sides of an ocean, (Teasley et al.; 2000, Herbsleb and Mockus; 2003). Since most software projects are distributed, either because the developers work in different locations or because customers are not co-located with the developers, effort has to be put into facil-

itating communication across the project. Where social networks exist across organisational and other boundaries there is more communication and project teams are better informed and coordinated, (Kraut and Streeter; 1995). Building a social network across sites is difficult. Team building exercises where staff from different sites come together can help as can working with one or two named individuals at the remote sites , (Kotlarsky and Oshri; 2005).

Distance can be broken down by the use of tools such as instant messaging which support ad hoc interaction and which use a conversational style of interaction, (Handel and Herbsleb; 2002). However the introduction of these types of tool does not mean that they will be successful. Herbsleb et al. (2002) found that there has to be a critical mass of users at each site before any communication technology is adopted usefully. Using the tool has an associated transactional cost, (Kraut and Streeter; 1995), and if few people are doing so the user is less likely to find the person or answers that they need.

3.7.3 Discussing technical issues

Learning new technologies, products and approaches is part of the life-long learning which all software developers expect. Good developers need to be aware of their shortcomings or of areas about which they need to learn more. Hoover and Oshineye (2010) describe two patterns that are helpful when encountering something new: *Expose Your Ignorance* and *Confront Your Ignorance*. Their book is a guide to *apprenticeship*, it describes many ways to learn to become a better programmer. Followers of the book form a community of practice around the idea that a developer can improve their skills and knowledge and around tools and techniques for doing so. Each approach in the book is

defined as a *pattern*, a common approach in modern computer science which builds on the approach used by Alexander (1978) in writing about architectural forms. Hoover and Oshineye (2010) give thirty-four different patterns which any programmer might follow as they strive to become better at their craft. Each is given a meaningful name, its context is described, the problem outlined and a solution described and demonstrated.

Expose Your ignorance

The context for this pattern is that those who are paying you to be a software developer are depending on you to know what you are doing.

It is not only employers or customers who rely on a programmer's expertise. Colleagues depend upon each other knowing what they are doing and doing it to the best of their ability. Assuming that one's colleagues know what they have to do and how to do it ought to be a safe bet. After all that is what they are paid for. However, in programming, developers constantly encounter new technologies and new problems and are expected to assimilate complex information quickly.

Confront Your ignorance

In this pattern a programmer realises they have gaps in knowledge or skills. Whilst some, perhaps all, of their colleagues possess the knowledge or skill the programmer may struggle to ask them for help.

Dweck (1986) studied the ways in which students approached failures in skill or knowledge-based tasks. She found that "many of the most accomplished students shied away from challenge and fell apart in the face of set-

backs. Many of the less skilled students seized challenges with relish and were energized by setbacks".

In defining these two patterns, Hoover and Oshineye (2010) are motivated by their solutions. In the first case they suggest that one asks questions, but recognize that "this is easier said than done, particularly when the person you're asking has assumed that you already know the answer". In the second case they suggest that programmers ask their mentors if anyone already has this skill and is willing to share what they know.

3.7.4 Shared perspectives within teams

If people communicate carefully and regularly then their ideas about the project and the product which they are building begin to align. The team will be more cohesive as they are "pulling together" in the same direction. However to achieve this state the team members need a shared perspective. "The problem of integration of knowledge in knowledge-intensive firms is not a problem of simply combining, sharing, or making data commonly available. It is a problem of perspective taking in which the unique thought worlds of different communities of knowing are made visible and accessible to each other", (Boland and Tenkasi; 1995). Collaborative tasks such as product design require that a team reach a state of shared perspective on aspects of the problem which they are solving, (Cross and Cross; 1995).

The shared perspective arises not only from foregrounded and shared information. Some of it comes from tacit knowledge which is constructed as people attempt sense-making activities, (D'Eredita and Barreto; 2006). Tacit knowledge can be developers' perspectives, their mental models of the world

or the concrete skills which they require in specific contexts. Perspectives are abstractions which become more solid, more meaningful as a result of activity and experience – as they are tested by experience.

Shared perspectives are an important aspect of coordination within a team, especially in a large team. A number of theories have been developed to explain how a group of individuals develop this type of communal understanding. Collective Mind Theory is one such explanation, (Weick and Roberts; 1993). "The major claim of collective mind theory is that individuals develop shared understandings of the group's tasks and of one another that facilitate group performance", (Crowston and Kammerer; 1998). For ethnomethodologists this process is always going to be one which is dynamic and which is never resolved because understanding changes as the group works together. Collective Mind Theory requires a that team members have a "disposition to heed", (Crowston and Kammerer; 1998), through which they behave in ways which foster the aims of the group. Such a disposition is developed through social interaction, (Kotlarsky and Oshri; 2005).

When team members share perspectives and context they begin to make assumptions about what their colleagues will do and plan their own actions accordingly, (Espinosa et al.; 2002). Stewart and Gosain (2006) suggest that one reason for the effectiveness of some Open Source software development projects is that members start from exactly this position of a common idea about what is required from the project.

3.7.5 Being a community

Workers are typically organised into formal groupings which give structure to an organisation. Organisation groupings such as development, sales or testing are used to define functional areas within which similar tasks are performed. Most project based organisations such as software houses have another, orthogonal, set of groupings based around individual projects or specific clients within which each member is nominally working toward the same goals. Most people are used to the idea that these types of structure can be drawn onto an organisational chart but each of us has any number of additional informal networks made of the people with whom we interact and which cannot easily be mapped. Such networks have been called "communities of practice", (Lave and Wenger; 1991, Wenger; 1998).

A community of practice is different to other sorts of community such as the people with whom we eat lunch or watch sport. A community of practice is based around some activity – the practice – something we do together and about which we learn by being engaged in it communally. Wenger suggests that a community of practice is instantiated through

- "sustained mutual relationships — harmonious or conflictual
- shared ways of engaging in doing things together
- local lore, shared stories, inside jokes, knowing laughter
- certain styles recognized as displaying membership".

These four ideas have a combined focus on social practices as ones which are productive of a working culture, but a community of practice does not have to be based in work or the workplace. It is a shared culture, but one

which is specifically shared around a *practice*. In reality work is made from myriad practices which together form a whole and it is the orientation to this set of practices which becomes the workplace culture.

The community aspects of workplaces differ, but so does the way in which people actually work. Even common tools such as Visual Studio can be applied in many different ways, they are, in fact, designed to be flexible and to not impose a style of usage. Within a team a common understanding will develop around how to use the tools, around what errors actually mean to them, even around how to structure or manage a codebase. Team members "engage with these practices in virtue of their place in the community of practice, and of the place of the community of practice in the larger social order", (Eckert; 2006).

Part of a community of practice is the development and sharing of collective knowledge and understanding. This may be knowledge about the project, the code and designs, the clients or the development techniques which are used. The knowledge of the team is a function of the people within it and of the social networks which they build, (Becks et al.; 2004). Within a community of practice members are engaged in mutual sense-making about their work and about the context within which they undertake it. They are making sense of their daily activities within the project, of their customers and of the products which they build, (Eckert; 2006).

The community's knowledge is a collective knowledge of successes and failures – what works and what doesn't, both individually and collectively. Some of this information is shared formally in meetings but more often it is shared informally. Staff can spend significant amounts of time talking informally, Herbsleb and Mockus (2003) suggest over an hour per day and one

informant in Chapter 5 suggested almost three hours. Yet a project such as Chandler, which was described in Section 2.4, (Rosenberg; 2007), or the computerisation of the London Ambulance Service, (Heath and Luff; 2000), can fail because the community lacks a historical memory of past problems and no knowledge of failures in other communities.

3.8 Empirical research into software engineering

Research in software engineering tends to study "better" ways to build "better" software, (Parnas; 1998). Software engineering's origins as a technical discipline have lead researchers to "a preference for quantitative research approaches that lend themselves to measuring causal relationships for successful software process improvement", (McLeod et al.; 2011). This preference, in turn, leads to a preference for those studies which create new designs or implementations over those which look at the people who make software, (Hevner et al.; 2004).

However, the increasing diversity and complexity of the processes involved in software development mean that research performed under the general category of "software engineering" is becoming more diverse. Within the discipline there is a growing interest in finding ways of understanding how software developers actually behave. Those behaviours can only be discovered by visiting and studying developers at their place of work as they undertake that work, (Sjoberg et al.; 2007).

When researchers study developers at work they are able to uncover how individuals and teams work and to reveal "human issues" within the discipline, (Seaman; 1999). Studying the people who develop software is not new,

perhaps because software engineering is a technical discipline there has been "a preference for quantitative research approaches that lend themselves to measuring causal relationships for successful software process improvement". Quantitative studies of software engineers, especially studies which are based around laboratory experiments tend to give an incomplete picture because the subject sits at the intersection of machine capabilities, human capabilities and human behaviours, (Seaman; 1999). Most researchers have worked on the first two of those areas, only relatively recently have researchers started to think about the third one. In particular the role of developers' behaviour has become a topic of interest and relevance, (Lethbridge et al.; 2005).

Researchers have, typically, struggled with the design and implementation of quantitative studies in software engineering. In particular experimental approaches work badly because the research must work with human subjects, typically sample sizes are too small to give statistically significant results, running experiments is expensive and revealing the context of the findings is difficult in an experiment, (Carver et al.; 2004).

Because qualitative and experimental studies fail to reveal the intricacies of software engineering practice some researchers are turning to qualitative studies including Seaman (1999), Lethbridge et al. (2005), Sjoberg et al. (2007), McLeod et al. (2011). Qualitative studies produce data "which are represented not by numbers but by words and pictures" and in which the complexity of the phenomena of behaviour are revealed rather than being abstracted away as it is in much quantitative work. This approach gives researchers the opportunity to create a "holistic" data set, which reveals "shared values, assumptions and beliefs, and the influence of particular individuals" within development

teams, (Sharp et al.; 2000, Robinson et al.; 2007).

The majority of research in computing, and certainly in software engineering, comes from a positivist tendency. Such research provides a way of producing knowledge which can be applied directly to the activities of developers without too much difficulty. Problems are classified as solvable and the positivist researcher works through a structured process toward the solution, (Lázaro and Marcos; 2005). If a solution is found, further studies may follow its application to "real world" problems as a way of providing some external validity. House (1970) suggests that researchers working in this way need to publicly demonstrate their results so that theories and laws are not formulated on the basis of weak hypotheses.

People act in ways which are subjective and contingent. Subjective research methods recognise the complexity of peoples' activities and reject the more explicit determinism of scientific experimentation. Subjective methods attempt to create *verstehen*, an understanding, of the sense-making in which people are engaged, (Abel; 1948). People's activities are seen as purposeful rather than being determined by external forces such as social structures or economic factors.

If software development is "engineering" then a common-sense understanding would suggest that it is surely based on rigour and on reproduceable processes. Authors such as Florman (1976), Petroski (1996), Molotch (2003) demonstrate repeatedly that engineering isn't like this, it is a human process which is often successful because of its unstructured nature. As with much engineering there are aspects of software development which are subjective or contingent. The choice of algorithm, for example, may not be based on neutral,

value-free factors. It is the result of a choice which is based on personal preferences, experience, the type of project and existing systems and code with which one must interact.

Empirical software engineering research reveals the realities of software development processes. When subjective methods such as ethnography are used to gather data the subtleties, nuances and complexities inherent in development processes are brought to the fore and become matters which are worthy of study in their own right.

3.8.1 Criticisms of empirical software engineering

Using qualitative methods does not provide a panacea for all of the criticisms of Carver et al.. In a meta-analysis of empirical studies of agile software development Dybå and Dingsoyr (2008) found that the studies were weak because "methods were not well described; issues of bias, validity, and reliability were not always addressed; and methods of data collection and analysis were often not explained well".

Even large studies are not immune to some of these criticisms. Studies which examined pair programming were found to "have not accounted for the moderating effect of the complexity of the programming tasks, which, in turn, may depend on the complexity of the system being developed or maintained and the expertise of the programmers", (Arisholm et al.; 2007). Basic flaws in the design of studies support the idea that qualitative studies are useful only for exploratory studies, are subject to bias and are inadequate for generalization. Such criticisms can be addressed through clarity in the research design and in its description, by providing a clear evidence chain and through

triangulation between studies. The difficulty which researchers have interpreting or generalizing their work on agile methods should not be. McLeod et al. (2011) state that software engineering is "a complex and intersubjective social reality that is interpreted rather than discovered". Qualitative, empirical studies of software engineering must work in this intersubjective reality to "uncover and elucidate problems and thereby delineate their solution spaces; challenge received views; and provide rich narrative accounts of practice", (Robinson et al.; 2007).

Longitudinal studies provide clear benefits over shorter ones. Processes are connected to outcomes in ways which are both unpredictable and emergent and which are only discernible through a fuller, temporal, immersion in a project. Researchers who are immersed in a project over time can discover events, meanings and rationales which might not be found through interviewing participants at the end of the project because of faulty memories or *post hoc* rationalisations. Indeed participants views, positions and ideas change during a project and it is only by being present that the researcher can see these changes and place them in context.

A genuinely holistic understanding of a project requires that the views of a range of participants are considered within the analysis. Relying too heavily on one perspective or a few sources can, perhaps unwittingly, introduce bias. Organisational policy documents such as corporate strategies, programming guides or Human Resources procedures are especially likely to give a limited perspective unless one is analysing corporate discourses. The "big picture" provides a useful context but may have little useful to say about software development as it is actually done "on the ground" by developers working with

clients. The experience of those developers can only be understood by talking to them about their work as they work, (Sharp et al.; 2000, McLeod et al.; 2011).

3.9 Ethnography and Software Engineering

There is a long history of cross-fertilisation between ethnography and software engineering. Typically this relationship is the use of ethnographic techniques as part of software development projects, there are fewer ethnographies of software engineering projects. This Section will briefly examine this relationship.

The problem that on software projects requirements are constantly changing and that this creates problems has long been recognized and there have been many different approaches to fixing requirements or otherwise solving the problem. One solution is to work closely with end-users so that the requirements and design documents closely reflect their needs, (Jarke et al.; 1999). This may involve embedding an outsider such as an ethnographer with the user community and with the developers, (Sommerville et al.; 1993, Twidale et al.; 1993). Techniques from outside the software engineering discipline such as ethnography can dramatically influence key aspects of the design of systems, particularly around the Human-Computer Interface, (Anderson; 1997).

The gap between technologists and ethnographers is far from insurmountable. In the last twenty years a large number of projects in engineering, medicine and the media have benefited from the presence of ethnographers, (Heath and Luff; 2000, Button; 2000). Many of these efforts built on the work of (Suchman; 1987) in working on the development of artificial intelligence for a photocopier.

Suchman's early work clearly showed that the data gathering techniques and analytical approaches of social scientists could be used to inform the design of technology in ways which benefited both developers and users.

The benefit to software developers of engaging with studies of the ways in which their customers work, or understand their work is clear. By better understanding the customer they are able to work to a more refined set of requirements. The detailed studies of ethnographers look beyond documented working practices or talk to examine the ways in which people actually work. They "serve as a foundation with which to consider how artefacts... feature in the production and co-ordination of social actions and activities", (Heath et al.; 2000).

The idea that an ethnographer might be able to help design software can still seem counter-intuitive to traditional software developers, but ethnographers have skills which ideally suit them to the task of understanding the needs of users, (Sommerville et al.; 1993, Hughes et al.; 1995). Ethnographers and computer programmers use different technical jargon and may perceive the same thing in different ways and may attribute different meanings to it. Sommerville et al. write of ethnographers that "although they produced insights into the organisation of work which are clearly of interest to systems designers, it is not clear how to translate their results into system requirements". The difficulty often is one of identifying material which is relevant to a system design within the detailed anecdotal and observational writing which characterizes ethnography.

Anderson (1997) argues that the social scientists who followed Suchman appeared to offer a methodology through which requirements capture might

be improved. They offered an approach using fieldwork and ethnography. The fieldwork would often be participant observation, whilst ethnography offered "a particular analytic strategy for assembling and interpreting the results of fieldwork". System analysts and requirements engineers have tended to be more interested in the phenomena seen in the field work than in the analysis provided in the ethnography.

Systems design is a social process built on the relationships users have with developers and that developers have with each other. When development is considered in the social realm rather then in the technological realm, it can be analyzed in different ways. Conceptual and practical similarities can be seen between foundational aspects of IT design and those of ethnomethodology. These similarities were called *technomethodology* by Dourish and Button (1998). Both software development and ethnomethodological analyses of social interactions share the need for *abstraction*, the creation of *accounts* and, ultimately, *accountability*.

Increasingly those developers with an interest in using ethnographic approaches to understand users are doing the fieldwork themselves rather than using a specialist. This work is "in danger of diluting the initial thrust of sociological studies of work for design purposes", Button (2000) because it too often concentrates on what is seen in the field rather than what is uncovered through an analysis of the data from the field. Rather than revealing the understanding of participants, or their methods for creating it, studies which lack an analytical framework become no more than "scenic fieldwork", (Button; 2000).

Whether rigorous or superficial, the engagement between software devel-

opers and social scientists is part of an important trend in development which foregrounds social concerns. These include not only the specific and situated concerns of users but also those of the developers who must deliver functioning programs on time and on budget. Agile methods, which address developers' concerns, were appearing at the same that Anderson and others were trying to merge ethnomethodological practices with those of software development. In introducing technomethodology, Dourish and Button (1998) make a persuasive case for a model of development in which an integration of approaches leads to better outcomes for developers, users and social scientists.

3.10 Summary

As this Chapter has demonstrated, ethnography provides a useful approach to understanding work. Software engineering is a specialised type of knowledge work whose practices are arcane and hidden yet whose artefacts are fundamental to so much of life. Researchers using ethnography are able to study the working practices of software developers as they strive to reconcile the needs of customers and the constraints imposed by colleagues and employers.

At the same time, software developers often work at the limits of their understanding of the tools, such as languages or libraries, which they must use and of their understanding of the customers' needs. Ethnomethodology provide an analytical framework through which the contingent and transient nature of these understandings can be revealed. Ethnomethodology reveals to the researcher how developers think about their understanding of the problems which they face and how they communicate those understandings with

their colleagues.

Although software development is most often realised through texts, both design documents and source code, the process is a social one in which ideas are shared and activities coordinated. Although many software tools are used to support them, understanding and coordination happen through developers talking to each other. Software development is always a socially constructed process in which the software which is designed and built is constrained by the relationships between the developers and users. Because so much software is built on top of existing code, developers expend significant time and effort in understanding that which already exists. But the meaning and structure of a system are not solely embedded in the source code, they are found in the design texts and diagrams. Understanding of the texts comes from developers talk to each other but is rarely clear cut. Rather the meaning of code and documentation is both negotiated within the project and dependent upon the context of both the production and use of those documents.

The creation and use of understanding by teams of developers is, clearly, a topic which is worthy of research. It builds on both ethnomethodological studies of work and the reflexive interests of software engineers. The next Chapter discusses the research methodology used to gather data and to analyse it in this work.

Methodology 4

4.1 Introduction

The previous Chapters have argued that software development is a negotiated and contingent activity and that the work of software engineers is a social process as well as a technical one. Developing an understanding of those social processes has to be based on data from the engineers' workplaces. A growing number of researchers in software engineering are using qualitative approaches to such studies. Much of this qualitative research was shown to have a tendency towards quantifying findings which could then be presented as "improving" the quality of processes, the productivity of developers or any of numerous aspects of development. Whilst quality or process improvements are important aims, this research is a more fundamental attempt to understand how developers' social practices reveal their understanding of problems and solution and show how they coordinate their work within the Agile Methods paradigm.

This Chapter is a discussion of the design and implementation of my field work, the data which were gathered during the fieldwork and the analytical techniques used in this research.

4.2 Principles which underpin the design of the study

This Section outlines why the study was designed as it was. It looks at the analytical goals of the study and identifies the data which are required for that analysis. These principles are placed within the context of other qualitative studies of software engineering.

This work is informed by an ethnomethodological epistemology. This leads to two defining characteristics of the research design. Firstly, that the ways in which developers understand their work will be found in their talk-in-interaction as they work and, secondly, that talk will be found in the workplace in real projects. Thus the relevant data for this study are the developer's talk and the context within which they are talking, (Garfinkel; 1996, Sjoberg et al.; 2007).

Talk in the software development workplace reveals the developers' understanding of the work they are each performing, their understanding of the work of colleagues and others and their use of tools, programming languages and design methods. Developers construct and share an understanding of existing source code and applications which they are using in their own work but which neither they nor their colleagues necessarily developed. All of these "understandings" may be shared within a team through documentation or, more likely, through talking about them. As the work is discussed, relationships between team members such as power hierarchies are embedded within the talk and can be revealed through the micro-analysis of talk using conversation analytic techniques and concepts.

The context within which work is performed affects both the work and the worker. A developer sitting in a cubicle maintaining code all day has a different work experience to someone who works as part of a team and pair pro-

grams in an open-plan environment. Workplaces each have their own unique culture. Even within a single organisation each team, office, production line or project will be unique in some way. Revealing the culture of the workplace through an analysis of the talk-in-interaction of employees reveals the context for the production of understanding. This gives an ethnomethodological orientation to the research.

The research questions here are to ask how agile developers share their understanding of their work and how they coordinate that work. Ethnomethodology provides a way of answering these questions through the accountable talk of the developers through which they both coordinate and make their knowledge indexical. By using techniques from conversation analysis, phenomena such as identity, face-management, politeness and the management of relationships within teams can be revealed in the developers talk to each other. These phenomena can be, in turn, indicative of coordination and shared understanding. But such phenomena have to be placed within the workplace and, specifically, within activities of work.

The study was designed and implemented so as to gather both recordings of conversations and wider background information about the work of developers and the activities taking place as the recordings were made. A number of principles were used in designing the study.

1. Acquire samples of developers in conversation about their work *as they work* using audio recordings of those conversations.
2. Use the rich descriptive power of ethnography to show the culture of workplaces and to provide a context for the talk.
3. Study developers in their place of work as they undertake their usual

commercial work.

4. Integrate the rich data of ethnography with detailed conversation analysis.
5. Gather data from a range of organisations which use agile practices.

4.3 On doing ethnography

Section 3.2 discussed why ethnography is at the heart of studies of cultures. Ethnography has been used to study many different aspects of work in many different working environments. This work is an ethnographic study of practising developers. This Section is a discussion of the use of fieldwork to gather data and of the writing of ethnographic accounts to reveal cultural practices.

4.3.1 Fieldwork

Fieldwork is the defining feature of anthropology. It was co-opted into other disciplines, initially in the social sciences and later into design, engineering, science and, latterly, software engineering, because of its appealing simplicity and the detailed data it can provide. Moeran (2006) gives four identifying characteristics of fieldwork:

- Intensive participant observation.
- The researcher must "be there", socially immersed.
- Where possible fieldwork should last for a long time.
- The research should develop an intimacy with the participants.

Pragmatic considerations of time and cost mitigate against these for many researchers, (Lethbridge et al.; 2005), but the basic idea that the researcher is

with the subjects for as long as possible, and becomes *as one* with them is an ideal to which fieldworkers aspire.

Participant observation requires that the researcher be present in the culture, immersed in its rhythms and developing relationships with its members, (Crang and Cook; 2007). Such observations are richer and more complex than those which might be made by an external observer because they include both the subjective, lived experience of the observer and their later, detached and more objective analyses. Since the researcher and the researched cannot be readily separated, Crang and Cook argue that participant observation leads to an *inter-subjective* understanding.

Fieldwork is potentially open-ended within the range of phenomena and number of willing informants the researcher finds. However few researchers have the luxury of following large numbers of potential leads in the pursuit of a comprehensive data set. Most fieldworkers have to limit themselves and their analyses to a few cases. A case study is an "empirical method aimed at investigating contemporary phenomena in their context", (Runeson and Höst; 2009). Case studies focus on the dynamics within a single setting, (Eisenhardt; 1989), often using several data sets and data types which can be triangulated to give increased validity to the findings.

Some researchers use data from a number of cases so that they can create a broader picture of the domain which they are studying. In uncovering the ways in which the live music of jazz is constructed by performers, (Faulkner and Brecker; 2009) used evidence from a large number of jazz groups they knew or in which they had played. Both Faulkner and Brecker were jazz musicians as well as sociologists who were immersed from an early age in the

world of small, working jazz bands. An outsider who knew little of jazz or of musicians would have needed to spend a lot of time with a few musicians to understand how they construct songs and sets. Faulkner and Brecker were able to use their experience as both musicians and sociologists to draw conclusions which were radically different to those an outsider might have reached. The breadth of experience they brought to their writing meant that those conclusions were richer then they might have drawn if they had looked at a single group.

Conversely, a deep immersion in a single case can reveal more details of the lived experience of work which might not be found when several cases are studied. When Moeran (2006) studied a Japanese advertising agency he immersed himself in the culture of the company over a long period of time. A more superficial view over a number of agencies would have said less about the culture of that industry in Japan.

Van Maanen (2011) writes about his early research into the culture of policing. For him the initial problem was one of access. To work alongside patrol officers he had to get access to the station and, despite having authorisation from senior officers, this access was controlled by station house sergeants. One of his most important early successes was in winning the trust of a sergeant who let him go out on patrol. Once he was patrolling with the Police, Van Maanen spent a long time becoming familiar with the work of a single station house, regularly patrolling with the same officers. Here, again, understanding comes from immersion.

Case studies can be, and sometimes are, criticised for being a limited basis for generalisation, that theory cannot be built on a single case and that there is

tendency to select cases which verify the research objectives, (Flyvbjerg; 2006). Theory, reliability and validity are all questioned by these objections, objections which would invalidate the use of such studies in all but preliminary work. Flyvbjerg demonstrates that such objections can be overcome with careful research design, a careful write-up and a nuanced interpretation of results.

Selecting cases requires care and attention. The cases which are chosen will affect the data which are gathered and the analysis which arises from those data. Research about social relationships is built from social relationships, (Crang and Cook; 2007). Studying a culture, obviously, requires that the researcher gets access to that culture. In workplace ethnographies access is not a simple matter, not least because finding a suitable "host" organisation is often difficult. Yates (2012) suggests that finding a company which is willing to act as the subject of software engineering research can be done through personal contact or the recommendations of companies or individuals who have previously worked with the research organisation.

Whatever method is used to find hosts, they will typically have three objections to, or difficulties with, the research agenda which must be addressed. Companies worry about their intellectual property, time pressures on their staff and specific worries about the research design. Each of the companies which co-operated in this study had these worries but they were eased by including the staff in the research design and taking time to talk them through the research process. In each company their intellectual property was protected through the use of signed non-disclosure agreements, staff were told that they only needed to co-operate when they had time and that they could end interventions at any moment and all personal data was to be anonymised.

The research design, specifically the ways in which data were gathered and used, was agreed with managers at each company and with the staff who would be participating. Negotiating each point with them meant that the key participants at each company became stakeholders in the research itself.

Having found a suitable and willing organisation the researcher must gain access to the specific areas which are interesting to them. Whether interested in the boardroom or the shop floor the researcher must negotiate organisational and corporate hierarchies in which issues of trust and confidentiality must be repeatedly addressed. Often access is controlled by "gatekeepers", (Crang and Cook; 2007), who will represent the researcher and their research more widely within the organisation. The gatekeeper is not necessarily the initial contact but is someone who has the power and influence to support the research agenda. The gatekeeper can impact implicitly upon the research. If the researcher is invited in and supported by management there is always a possibility that they will be seen as management's stooge or spy whereas if the gatekeeper is one of the workers or a trade union official the researcher may be mistrusted by the management, (Gill and Johnson; 2002).

The process of finding subjects for the studies which are presented in this thesis is described in Section 4.4.1. The negotiation of access to those companies is described in Section 4.6.

4.3.2 Field notes

Recording that which is seen, heard, felt, understood or mis-understood whilst in the field must be done by taking notes. Those notes may be contemporaneous or written some time later, they may be raw or refined through editing

but they must be treated honestly because they are the raw data of the study on which the analysis will be based.

The fieldworker cannot note everything which happens and must either make selective notes or create a broadly impressionistic set of observations. Crang and Cook (2007) write that noting participant observations is difficult and that "a considerable amount of field noting gets devoted to 'self-reflections'". In unfamiliar situations researchers can turn to the one thing which they are most familiar, themselves, and make notes about it. If too much self-reflection can be avoided, the researcher must find phenomena which are both interesting and which may help to answer the research questions and these may not be phenomena which were identified *a priori*.

Section 5.5 includes an account of a Product Manager at Z* publicly berating a developer and demanding that he be available to talk to their client later that day. This was noted because the behaviour was atypical of the way in which Z* worked before they made their agile move, but also because key informants had said that communication around their nteraction with clients was difficult. Later, once the company was using Scrum, the same Product Manager was cooperative and collegiate in a similar situation. Although the fieldwork was not about client interaction, notes on these changes helped to reveal cultural changes across the company in their organisation of work.

4.3.3 Writing ethnography

The creation of an account, especially one based on close observation in the field, of a culture is not simple: it happens in the writing process and, at the same time, determines the nature of that process, (Van Maanen; 2011).

When writing the ethnography, the fieldworker creates a representation of the culture but the conversion from field notes to ethnography is not a straightforward objective process. Van Maanen writes that the ethnography is mediated and transformed by the rhetorical and narrative approach which is taken in the writing process. Accounts of cultures can be broadly categorised into at least three forms:

- Realist tales are matter-of-fact descriptions of the culture.
- Confessional tales focus on the position of the fieldworker within the host culture.
- Impressionist tales produce more dramatic personalised accounts which mix both realist and confessional styles.

The differences between these broad classes can be subtle and may not matter except to other ethnographers. The tale which is written is an interpretation of the culture, the style of writing provides a way of performing that interpretation, (Van Maanen; 2011).

Regardless of the narrative approach the writing is an interpretation of the culture from the theoretical position of the researcher. Before the ethnography is written the field data must be prepared and a theory generated. Data can be prepared in a variety of ways.

One popular approach is coding. During coding raw data, such as transcripts of conversations, are analysed, contextualised and classified. Concepts are extracted from the processed data and made available for statistical analysis. Coding must be done carefully since the codes which are chosen influence what the data reveals in later analysis, (Lethbridge et al.; 2005). The chosen coding scheme must reflect the goals of the research project.

Ethnographic field notes lend themselves to this type of coding but the question of what is lost in the coding process has to be considered. By removing text from its wider context some of the richness and variety of that text is lost. On the other hand, coded field notes or interviews can be used in statistical analyses alongside surveys and experimental data, (Crang and Cook; 2007). The numerical analysis of such data can be added to an ethnographic account with the ethnography adding context and meaning which helps the reader interpret the numerical data.

4.4 Implementing the study

The principles which were identified in Section 4.2 were operationalised as the basis of the design of this study. The study would be based on a number of cases which were selected so that the whole set would reveal a range of phenomena.

Scrum is probably the most well known and widely used agile method. Scrum teams were more likely to be available than, for example, teams using DSDM or Crystal and approaches such as XP remain quite niche even within the agile world. Consequently the main thrust of the study was to be *within* Scrum teams whilst not being a study specifically *of* Scrum teams.

The structure of Scrum meant that some *a priori* assumptions could be made about where and when phenomena would be found. The planning meeting, retrospectives and daily stand-up are all meetings in which team members coordinate their work. Stand-ups provide a space in which the developers' work is discussed and sprint reviews are an opportunity to discuss the code which the whole team has produced. The stand-up and review seemed

to be situations in which the answers to this study's research questions might be found.

Scrum isn't the only agile method which might be helpful. Although few companies use the whole of XP, pair-programming is something which a number of companies do. XP is interesting here because it is, perhaps uniquely, a practice that is explicitly intended to support the sharing of understanding about code.

The foundation of the study would be to find and examine companies in which stand-up meetings, periodic reviews and pairing were practised. It was not likely that a team using Scrum would also pair since Scrum and pairing come from different approaches to agility. Therefore a number of different cases would be required with the analysis synthesising meaning from across the data sets.

At the outset of this study it was not clear if developers would talk with more detail about aspects of programming such as source code, design or testing in formal meetings or in less structured settings. One might surmise that less detail is given in formal meetings but this is not something which has been shown empirically when looking at software engineers. Detail and precision are likely to matter to programmers as they manipulate design and code which are both infinitely malleable. Uncovering talk in different situations to understand how the detail of code is made a matter for discussion meant that a number of situations had to be examined.

Two types of data were to be gathered. Rich descriptions of working environments and practices would come from field notes and unstructured interviews. Recordings of conversations between participants would be used to

create detailed analyses of their understanding of their own work.

This Section will examine the design of the field work, how suitable cases were found and how the results would be analysed.

4.4.1 Finding cases

Finding suitable companies which met the criteria I discussed earlier, was one of the major early challenges in this study. The companies which I approached had to be ones which were primarily involved in the development of software, which were using agile practices and which were sufficiently welcoming so as to allow me to observe their teams.

I was keen to understand how developers used talk about their work as programmers to construct different agile practices. If the only data points I gathered were from Scrum stand-up meetings my analysis would only be about that type of meeting. That is a worthwhile aim in itself but that was not my research agenda. I wanted to explore working practices across the agile community and to understand how developers use talk to reveal and share understanding and for coordination.

Therefore the cases had to be of agile companies and across the set of cases I was looking for there needed to be a range of agile approaches in use. I had a number of sources which I could use to find suitable companies. The research centre in which I am a member, the CCRC, had and has numerous industrial links. I could use personal contacts and was able to draw on more distant contacts through social media. Finally I had a large pool of current and former students who had, between them, a wide range of possible contacts.

I designed a small flyer which outlined the aims of the study and gave my

contact details. I emailed the flyer to my contacts and to some of my former students. The flyer gave the URL of a Website which had more details about me, gave detailed aims of the study, listed the characteristics of appropriate subjects and described how the research would be conducted and how the findings would be disseminated. The flyer and pages from the Website are included in Appendix B.

Before my advertising was ready the CCRC was approached by a local company who wanted help in making the transition to Scrum. Z* had worked with CCRC on a number of projects in the past and wanted the Centre's help in evaluating the impact of Scrum on their processes and products. Although their needs were rather different to my aim there was sufficient overlap to mean that this could act as a pilot study. Senior staff at Z* wanted to make a move to agile methods to improve communication within their teams. This was a good opportunity for me to see some of the ways in which developers, managers and possibly customers communicate about products and processes.

The flyer and Website produced a number of responses – all of them generated from former students. The companies which responded all identified themselves as using agile methods or practices and as being enthusiastic about agile. I emailed and talked to a number of companies before having detailed talks with the two which were most enthusiastic. I selected two companies because, I anticipated that when the pilot study was factored in, three studies would produce more data than I could reasonably analyse in the time which I had available.

Z* had heard and read lots about the benefits which agile methods are of-

ten said to bring and felt that by making an agile move they would overcome internal problems. A* and E* were companies which had been using agile methods for significant lengths of time and which used them for all of their development and deployment activities. They were keen evangelists for the approach with both managers and developers who were active in local agile groups. These set of companies with which I worked provided a contrast between Z* who were making a transition into agile, and the others who were enthusiastic proponents of the agile approach.

4.4.2 Designing and doing the field work

The largest quantity of data was the audio recordings of meetings and of developers talking together. The meetings were chosen in collaboration with the subjects because I preferred to record a meeting at which I was a welcome presence than make a fuss to get in to a meeting at which I was unwelcome. The samples of talk about code were gathered by sitting with teams as they worked and asking if they minded if I recorded their conversation.

In both situations I recorded only if none of the participants objected. I chose to record only where and when I was welcome because I had been invited into these offices as a guest and I had no wish to offend or upset anyone. But also I owed a duty of care to the people I was observing. The ethical concerns raised by observing people at work, and my approach to dealing with them, including getting explicit permissions, are outlined in Section 4.5.

I was aware that any observation can fall foul of the "Observer's Paradox", (Shanmuganathan; 2005), in which the act of observing changes that which is observed. When people know they are being watched or recorded they will

change their behaviour so as to maintain face or status. If I had forced my way into a meeting, the attendees would have acted differently because I was there and because I was an unwelcome presence. In this case I intended to be with teams over a period of days or weeks which was long enough to be accepted and meant that for most of my field work the participants would behave naturally. A* employed a number of staff whom I had taught when they were at University who behaved very similarly at work to the way they'd been as undergraduates some years before. Having seen them in action I was confident in their behaviour being natural.

I was also aware, in designing this study, that my status as a fieldworker had to be negotiated with the participants. Was I simply an observer or was I in some way going to be a participant in the conversations? The developers would know that I am a software engineer as well as a researcher. Part of the process of negotiating access to the companies would be to talk about their work, their processes and their clients so that I could be sure that they were suitable. As part of such conversations my status would naturally become clear. I intended to be an observer where possible and to try to minimise my participation in discussions about code. This was, in the end, easily achieved in meetings because of their structure but a little harder to achieve when watching people program.

My original research plan had called for video recordings. Büscher (2005) outlines the utility of video as a tool for fieldwork. Video is a powerful tool because, as Büscher shows, it allows for "[r]epeat viewing, 'dissection' through slow motion and frame-by-frame analysis". Events which may pass the fieldworker by unregarded may be revealed because they are part of a video record-

ing. Where collections of videos are made across projects or across studies, comparison can be made between times and situations in ways which are much more difficult when using field notes. However, I decided to use just audio because capturing this is far easier than making a video recording and would be less disruptive of the environments in which I was working. (Büscher; 2005) did not have the worry of disrupting the work of her respondents because they were engaged in a collaboration with her about their work as landscape architects.

The process of making an audio recording can be very discrete and is almost completely unobtrusive in many office settings. I had planned to use a high-quality audio recorder with a built-in microphone, and started to do so, but the sound quality was poor and the machine was quite large. Through a mixture of experimentation and necessity I found that my phone produced recordings of similar quality but with greater discretion.

Ethnographic data were drawn from contemporaneous field notes. Descriptions of background, of the environment and of context were noted down as they occurred to me with the vast majority of the field notes made as the developers worked. My approach was to sit with them and watch them work, making notes about their activities including conversations and their use of tools, navigation of source code and the other practices of the working developer. Occasionally I would write down my feelings and interpretations of the events as I saw them, but when I did so I would annotate the notes so that when reading back I could be clear about where observation ended and interpretation began.

I did not use structured interviews with people at the companies. From

the beginning I wanted to avoid being constrained in the ways in which we interacted whilst being free to move around gathering data as I saw fit at the time. However I was able to talk to people at all levels throughout my visits and wanted to record these informal conversations. Although I preferred to make notes whilst talking to people this was not always possible and in those cases I made a point of writing my notes as soon as possible after the conversations. Often, I found, conversations would take on the tenor of unstructured interviews since I intended to try to gather information about development practices wherever and whenever possible. I would let these conversations flow but would try to guide the respondent along lines which were suggested by my research agenda. In these conversations I was looking for detail about the context, information on projects and working practices which might be known to members but not immediately or easily uncovered by an outsider. Important contextual information was revealed in some of these conversations, particularly at Z*, which showed the tensions within the company about the working practices which they were trying to change.

These types of conversation provided me with pointers towards phenomena which I could study in more detail. For example at Z* the details of tensions between the customer-focused Product Managers and the software developers became a matter of interest because almost everyone I spoke to talked about them without prompting. In analysing the field notes and recordings I was, at times, guided through the data because aspects of these conversations highlighted matters which were important to the participants.

I made multiple visits to each company over a number of weeks. I tried to gather data which were representative of their work on projects even though I

would not be able to be present in the company all of the time. In fact, given a full-time work commitment I visited each company for only one day per week. The visits typically started at around 10 a.m. by which time people had dealt with emails and so on and were settling in to their days' work. I was usually present until the middle of the afternoon at which point the developers were generally engaged in coding rather than in talking. This schedule came from my early visits to Z* and mirrored the pattern of work there. With small daily alterations it worked well at E*, too. Working with two of the companies, Z* and E*, required visits over a number of weeks whilst A* provided enough data in a single visit. The number of visits and their scope was something which I was prepared to adjust as the work went on and which I negotiated with the companies.

The field work began with one or more preliminary visits to the company to discuss the rationale for the research, how data would be gathered, what my expectations of the participants were and to answer any questions which they might have. During these sessions I built a rapport with the developers and managers that I would be observing. At A* this was an easy task because I taught two of the developers when they were undergraduates. At both Z* and E* the process was a little more difficult. At both of these companies I went and met staff some days before my fieldwork started. We talked about programming and, especially, the tools and technologies which they used or had used on previous projects.

Meeting the developers who would become the subjects at the start of the visits provided an opportunity for us to get to know something about each other. I was able to informally discover things about their processes and tech-

nologies such as what they used, what they liked, what worked for them. In some cases I found out where they bent the rules or tested the limits of methods or technologies to make them work as required within a specific situation. In these early conversations I naturally revealed things about my own knowledge and experience. The developers I met were interested in what I knew about programming, what I might be able to ascertain about their work and whether I could validate their approaches. This sharing of experience was something I previously found when visiting companies to provide training or consultation through the University.

These preliminary meetings were also a good chance to look at work spaces, to see the layout of the offices and specifically the desks. Knowing how *their* work was arranged physically gave me the scope to redesign any aspects of my work that needed to change. For example, where the workstations were too small for me to sit with them around their work station I thought about how I would sit and observe and where the recorder would have to be placed.

4.4.3 Taking notes

Each of the case studies in this research uses ethnography. One presents purely ethnographic data whilst for the other two ethnographies provide contextual information which is used in applied conversation analysis. Whilst it is accurate to write that contemporaneous field notes were taken during each visit, this does not convey the difficulty which taking those notes presented. The physical act of note-taking was straightforward, unlike the situations described in Crang and Cook (2007) where, at times, researchers found taking notes to be culturally unacceptable. In this study the difficulty was in knowing *what* to

note. A multi-level approach developed across the visits.

During early visits to each company I would make general notes about their organisation, their products and working practices. Typically these would come from conversations with developers and managers as I was introduced to staff and shown around the office. Once I had been shown around I would be left to my own devices and would begin by sitting quietly and observing the way in which everyone worked. At this stage those general notes were supplemented with detail about the working environment, atmosphere and my feelings.

Based on what I saw and the initial introductions, I would develop a list of key informants. These were either people whose role was related to the organisation of work such as project managers, or people who were heavily involved in the work of programming. I would talk to these informants, making notes but not taking audio recordings. Often these conversations would lead to other informants or I would be told about meetings which I ought to try to attend.

The final level of field notes were those which arose "naturally" during my observations. When I saw or heard events which I felt related to my research questions I would take notes about them, noting the time, participants and as much detail as I could about the event.

After each visit I spent time reading through the notes, cross-referencing and identifying points on which I wished to follow-up during later visits. It was through this cross-referencing that I was able to construct the series of events which are presented in Chapter 7. Without the detailed notes and cross-references those interactions might have remained as separate items rather

than being identified as parts of the same piece of work.

4.4.4 Transcribing the recordings

Many empirical software engineering studies use a mix of quantitative and qualitative techniques. In these studies qualitative data is often prepared for numerical analysis using a process of coding. During coding raw data, such as transcripts of conversations, are analysed, contextualised and classified. Concepts are extracted from the processed data and made available for statistical analysis, (Lethbridge et al.; 2005).

In designing this study I wanted to work with purely qualitative data. In part this was to avoid the engineer's tendency to expect a "right" answer but it was also because I wanted to use the expressive and descriptive power which comes through the rich text of ethnography.

The initial study at Z* produced only field notes. The study at A* produced a recording of a Skype call in audio WAV format. At E* I initially used a dedicated audio recorder which produced files in a format called WMA. The recorder was so bulky and cumbersome that after two days I switched to making recordings using my mobile phone which saved the files in a format called AMR.

The process of transcribing the recordings from A* and E* was straightforward. I used audio conversion software to convert each file into MP3 format. MP3 is a "lossy" format which uses complex algorithms to compress the audio stream. The benefit of this format is that the files it produces are small and can be moved around easily. However in the process of compressing the audio some of the fidelity of the original recordings is lost, primarily in the

lower and higher frequency ranges. My field recordings didn't contain data in those ranges and the loss of fidelity was not a problem – had it been so I would have been able to work from the much larger original files.

I used a software application called Audacity to perform the transcriptions. Audacity is a sound recording and manipulation application which I used to slow the recordings, to navigate through them and to repeat sections. some of the files from E* had significant background noise. I was able to use Audacity's features to reduce the level of the background noise without impacting negatively on the foreground talk.

The talk was to be transcribed using an annotation scheme which exposed its structure and which is given in detail in Jefferson (2002) and, briefly in Appendix A. This transcription scheme is rich enough to reveal the structure of the talk: its sequential ordering and the ways in which this is managed by participants. In particular, it is the "context-specific use of rules, procedures and conventions" which are revealed, (Heap; 1997). Jefferson's transcription scheme lets the analyst show the detail of the production of order and meaning within a conversation without obscuring the detail of *what* is being said. The transcription scheme used here is a subset of Jefferson's as used throughout Ten Have (2007).

4.4.5 Answering the questions

The transcriptions and field notes were gathered so as to answer the research questions. Reading and cross-referencing the notes pointed to recordings which were transcribed. The notes showed events which were, in ethnomethodological terms, indexical of relationships or working practices. Listening to

the recordings of those events revealed the details of the ways in which the developers managed their identities and relationships. Reading the detailed transcriptions showed how the developers managed their face, how humour helped to bond teams and how their technical talk showed them to be communities of practice.

4.5 Ethical considerations

Fieldwork and its results are bound up with issues of knowledge and power and can become highly political. The status of the fieldworker is always likely to be contested. Section 4.3.1 briefly mentioned the possibility of the fieldworker being seen as a management spy or a union stooge, for example. Whilst the self-reflexive fieldworker will be aware that their position is not a neutral one they need also to be wary of over-compensating in a potentially futile search for objectivity. An ethnography is going to take a position on matters of power, hierarchies and relationships but in so doing the researcher must be careful to avoid misrepresentation.

A sensitive approach to power relationships is to work "with" not "on" the subjects, to try to work from different perspectives so as to get insights into different views, to triangulate data and theories using multiple data sources, (Crang and Cook; 2007). These goals can also be achieved by getting properly informed consent from participants and by carefully sharing data with the participants, (Yates; 2012).

This study had a number of potential ethical challenges. The primary concern was for the work not the participants. Individual developers could find their relationships with colleagues or with their employers made more dif-

ficult or, at worst, badly damaged if personal information was revealed or if negative attitudes were attributed to them.

The companies which were studied were at risk of damage to their reputation if they were shown to be somehow unprofessional or untrustworthy. By opening their practices and source code to a researcher they faced the possibility of losing control of their intellectual property.

Each of these risks is significant and had to be considered in the design of the study. A number of things were done to ameliorate the possible risks. Before I tried to find companies which were willing to help I had to get the approval of the design of the study from the Univerity's Ethics Committee. In asking for approval I wrote:

> *At all stages during analysis and publication data will be anonymised. Raw data such as field notes or video will be safeguarded by keeping them securely within the University.*
>
> *Companies will be protected from commercially damaging revelations because I will sign non-disclosure agreements with them in the same way that I would for consultancy work. Management at each organisation will be briefed on the details of the observations I intend to make, the data analysis and the methods of dissemination of results. I will make clear that all data will be anonymised.*
>
> *Individual developers will be given the same information but with a focus upon their personal anonymity at all stages. They will be asked to sign a consent form before observations begin. Whilst the work requires access to project teams during their daily work, some people may not wish to be involved. When I first introduce the work to them every potential*

participant will be told that they can withdraw at any time. The right to withdraw will be clearly stated on the consent forms.

In line with University guidelines participants were asked to give explicit consent to my presence and were told that they had up to one week to withdraw from the study after the completion of fieldwork. The consent form is included in Appendix C.

Approval was given for this study.

4.6 The three cases

Having discussed some of the ways in which qualitative researchers in software engineering and other domains gather and analyse data, the design of this research study can now be described. The data, data gathering approach and analytical method are chosen to answer specific questions. The research problem here is to understand the social construction of the working practices of software engineers. In so doing this research will interrogate the Agile Manifesto's commitment to favour "Individuals and interactions over processes and tools", (Beck et al.; 2001), by focusing on the social interactions of agile developers.

Fieldwork for this project was carried out at three points during the period 2008-10. Each intervention was at a different software house and looked at a different aspect of the introduction or use of agile methods. Although the companies and activities were different at each stage of the work, similar data gathering techniques were used throughout. Data was gathered in three ways: observations, unstructured interviews and audio recordings. The first two

techniques were used in all three cases but in the first only observations and interviews were used.

Software projects are often too long and too complex to be studied in detail. Few research projects, most certainly not this one, have the luxury of time, space and access given to Rosenberg (2007) in his study of the Chandler project. A more typical approach is to create targeted interventions which look at key moments in the life of a project and to support those with a longer term, but less intensive, ethnography. That is the approach used here. A small set of agile interactions which foreground communication practices will be examined. Specifically, the daily stand-up meeting of Scrum, end of Scrum retrospectives and XP-style pair-programming sessions. Each of these is a constrained situation which, their proponents would claim, developers can and do talk openly and in detail about their work.

The first case is a study of an organisation which is transitioning to agile methods from a highly structured process. The second case is a study of a Scrum team during their daily stand-up meeting. The third case follows two programmers as they work in a pair. These three case studies give broad coverage across the range of agile experiences which a developer might have.

The agile move, the introduction of agile into a team, is always going to be a disruptive process because it brings a change of culture to an organisation alongside a change of working practice. The first case study has the potential to reveal the pressures which can both drive and resist the introduction of agile into a team. The other two cases are studies of established agile teams working in organisations which are very supportive of the agile ideal. The second case study looks at a single event in the life of a Scrum. It is an in-depth analysis of a

stand-up meeting. This second case should provide opportunities to examine how developers become accountable to the idea of Scrum once they commit to using it. This accountability is something which might be expected at the first organisation if their introduction of Scrum leads to similar cultural changes. The third case study follows a pair of developers over a number of days as they work to understand some legacy code, write unit tests for it and create new code which uses it.

The three cases move from the broad detail of an organisation through an examination of how a small team works and end by looking at the detail of how individual developers make sense of their work as they write code. Each case will help to answer different aspects of the research questions. The first case, Z*, will show how the coordination of work changes as Scrum is introduced. The A* study will show the detail of the immediate, contingent, daily coordination of work within a Scrum and how that coordination is linked to shared understanding of design and code. The final case, E*, will be narrowly interested in the sharing of understanding as code is written, looking at the minute-by-minute development of meaning within a pair.

Together these cases will show that communication between developers is an integral part of the process of working in a software development team. Whilst the context for this communication is the use of agile methods the cases may reveal practices which do not rely upon agility for their utility but which, instead, agile methods must rely on for their success.

4.6.1 Z*

In late 2007 the Communications and Computing Research Centre at SHU was approached by a local software house who wanted advise on, and support to change their approach to project management. CCRC was unable to help them but colleagues and I met with their senior development staff for a brainstorming session. That meeting proposed a number of ways in which we could work together in other areas than project management. As a result they agreed to permit some observations as part of my research.

This company is a small software house which builds tools and platforms that are used in the production of DVDs and other digital data formats. At the time I began working with them they were running a structured waterfall approach to the Section 2.3. Their projects were often small pieces of work which lasted between three and six weeks delivering custom templates for DVDs. Most of the code was written using Google's GWT tool set and transformed into JavaScript and dynamic HTML.

The custom nature of the code meant that there were lots of interactions between developers, the sales team and their clients but the clients were usually based in the USA which created a range of communication difficulties because of differences in time zones. The short timescales which these projects faced meant that, inevitably, time for testing and final quality auditing was always compressed and the end period of projects was often fraught and stressful. Combined together these problems caused constant disagreements within the company.

The arguments made for the use of agile methods include process improvements which lead to better products as all parts of the team work together

within a single iterative structure. Section 2.5 describes the rationale for the idea of agility and some popular agile approaches. A number of books and conference presentations had convinced some of the development managers at Z* that switching to Scrum might help ease their problems and reduce internal conflicts. They didn't have enough support from either the Board of Directors or the developers to switch their entire process to Scrum but were to be allowed to run a pilot project. My role was to be to study how they worked both before and after the change.

In designing the study at Z* my key informant was one of the development managers. John initiated their Scrum trial and was to be closely involved in running it. My work, and my role as observer and evaluator, were delicate because I was invited into the company by the management and could be perceived as "their man" by the other staff. At the outset of the work it was possible that this attempt to reduce conflict could, in fact, cause it. John's role was crucial here. Although he had a managerial role as a lead developer, he still designed, wrote and tested code every day. Far from being detached from the development staff, he was one of them and, in pushing for the use of Scrum he was, several told me, expressing a widely-held desire. The only difficulty I experienced in my time at Z* was to be with a project manager who was fairly forceful in rejecting the idea of Agile methods.

Because the move to Scrum was politically sensitive with some staff at Z*, John and I, working with other managers, decided that the process I used would be quite lightweight. There was insufficient support for an intervention which took developers away from their work and reduced productivity, for example through lengthy structured interviews. The company and I agreed that

I could observe them, taking whatever notes I needed to, and talk to anyone who was willing to talk to me but only whilst they carried on working.

The process at Z* became one which would look familiar to classic anthropologists. I spent time in the office observing each piece of the development process. During these observations I found staff in areas from development, testing, QA and sales who became my key informants and who would be the subjects of unstructured interviews. The key informants were people who were involved in the troubled projects and who I was told by managers would be part of the Scrum project when it went live. These included developers, testers and the Product Managers. Notably the Project Manager was not involved in the Scrum trial either as a participant or as an evaluator.

Some of my informants had supervisory or managerial roles but others were regular developers of testers with no such responsibilities. The selection of informants could have influenced the results of this study by introducing unintentional bias. Managers may have had a different view of work to those who were actually doing it, those who were promoting Scrum would have given different information to that which was given by those were opposing it. In this work I was given free rein to talk to everyone and anyone and was able to select my own informants which meant I was able to be proactive in searching for informants who played key roles in, or who held strong opinions about, the changes.

The selection of informants arose from a process of unmotivated searching. I was not looking for specific phenomena and deliberately tried to avoid having pre-conceptions about what I wanted to see or hear. Instead I watched people at work, talked to a large number as they worked and chose to spend

longer with those who gave me more time or more information. Although I had been brought in to the company to evaluate thee use of Scrum I was not there as an advocate of the approach – they brought in a consultant to fill that role. In fact, at the time that I started working at Z* I would have categorised my feelings about Scrum as being that it was insufficiently agile and that it was overly structured. As I moved around the company I was able to gather a wide range of views on their problems and on potential solutions. The key informants became those whose views contributed to a rich picture of Z*.

More details about Z* and its products are given in Chapter 5.

4.6.2 A*.com

I found A*.com through a former student. I had asked a number of colleagues and on email lists for companies who would be willing to let me observe their processes without success. I decided to search more widely by creating a Web site which would advertise my work and ask for participant organisations. The Web page found its way to one of my former students who contacted me to say that the company he worked for, A*.com, would "probably" give me some access.

A*.com was another small software house. They specialised in the secure online storage of documents such as insurance quotes and policies, bank details, wills and so forth. When originally established they had tried to sell their service direct to consumers but, at the time of my study, they were marketing it to banks and insurance companies instead. A* were interesting because of their software architecture. The code was split into two areas: back-end services and infrastructure; and Web clients. The infrastructure code turned out

to be large and complex and in a state of change which was forcing some major redesign and rewriting.

Whereas Z* were trying to become agile A*.com was a fully-fledged agile shop. They used Scrum fully, engaging with all stages of the Scrum cycle. Having to manage and modify a problem codebase is a textbook situation for Scrum. It requires the kind of iterative thinking, implementing and testing for which Scrum is designed and is probably best done with the support of automated version control, a backlog and a test-driven process. A*.com used all of these.

To gain access I emailed back and forth with my contact and his managers. They were happy to have me go in provided that I didn't reveal anything which was commercially sensitive. Commercial sensitivity applied in all three cases but only Z* asked me to sign a confidentiality agreement: the other two companies were happy with a verbal agreement and a handshake. I discussed the best approach with the developers. I wanted to attend one of the key Scrum meetings but had only a limited window of availability. At the times I could be there they were mid-Scrum and the only meeting they were having was the daily stand-up. I decided to attend one of these.

The stand-up meetings at A*.com were slightly different to the classic model as described by Schwaber (1995) and which I saw in action at Z*. Most of A*.com's developers were based in Sheffield but the lead developer, who acted as Scrum Master, lived and worked in Stevenage. The stand-up meetings took place as Skype conference calls which meant that everyone would normally attend even if they were not in the office. It also meant that I was able to make an audio recording of the whole call using a piece of software called *Skype Call*

Recorder.

I arrived at A*.com at about 08:30, some time before the morning call. I talked to the team, it transpired that I taught two of them when they were undergraduates, and was shown around the office. In this study I was more interested in gathering information about their product and processes than in richer ethnographic detail. I talked to the team about the way that they worked, in particular about their use of Scrum and about the work which they were undertaking.

The recording of the call went smoothly. I was silent during the call even when questions and interventions suggested themselves to me. My purpose was to record their talk during their normal interactions which would not have been possible had I intervened.

The recording was transcribed using a conventional set of Conversation Analysis annotations which are given in Jefferson (2002) and included here as Appendix A. The annotated transcription was analysed first through repeated reading to reveal potentially interesting features, then in a series of guided and detailed close readings. The initial reading served to filter out sections of talk to reduce the volume of data which would be examined in detail. As features of the transcriptions were revealed they were shown to demonstrate aspects of key ethnomethodological concepts including accountability, reflexivity and indexicality. These ethnomethodolgical ideas revealed how the team were using the stand-up to manage status, coordinate their work and build coherence within the team.

More details about A*.com and its products are given in Chapter 6.

4.6.3 E*

The third case was a company in Nottingham. Again, I found them through my Web site and a former student who introduced me to the Managing Director, Adam. My initial contact was a series of emails and phone calls with Adam before I went down to Nottingham to meet him. By the time that we met he already knew that I was interested in recording and observing developers as they programmed and he was happy to provide me with the access I needed.

Adam was clear that E* was an agile company. He was proud of this, telling me that it was an important part of the vision for the way that the company ran. When they recruited developers to E* they looked for a mix of technical skills and agile experience. They had tried recruiting developers who were technically adept but "not agile" but, according to Adam, this tended not to work out because of cultural mismatches. Even developers who wanted to manage their own work would struggle with the mix of approaches which E* used.

Adam told me that, rather than follow a specific methodology such as Scrum, E* preferred to cherry pick individual agile practices. They used Kanban boards, estimation, backlogs and test-driven development but the defining characteristic of their development process was the extensive use of pair-programming.

E* appeared to be a perfect company for my research. They were open and friendly, access would be as unrestricted as it could be *and* they paired. Compared to Scrum, a process which centres on pairing can be unstructured, possibly even chaotic. Adam said that the freedom to pair was simply too un-

conventional for many experienced programmers and that he now preferred to recruit those who were younger and less experienced.

It was clear right from my first visit to E* that I wanted to watch pair programming and that I would need to make audio recordings which I could use for conversation analysis. I spent two or three days a week at E* over a number of weeks watching how they worked.

The programmers decided if and how they were going to pair. Conventionally, if anything in pair programming could be said to be conventional, developers pair in a formal way as driver and navigator and work together for days. Often the two people who pair have different levels of experience so that one is mentoring the other as they work. At E* none of this applied. I saw that people could work on their own if they wished but that, often, most of them preferred to pair. The would talk to each other to find a pair, sometimes collaborating for a few minutes to work on a tricky problem, sometimes spending all day together.

On each visit I would sit with whoever happened to be pairing and watch them work. I made notes throughout and if they were doing something which particularly interested me I would make an audio recording. My interest might be piqued by the nature of the problem, by the way in which they were talking or because their work was related to something which I had previously recorded.

By the time I stopped visiting E* I had pages of notes and many hours of recordings. I had far too much audio to transcribe or even to fully listen to. I read the field notes looking for areas which I wanted to transcribe. Having found suitable data, I created transcriptions with detailed conversation anal-

ysis annotations for later analysis.

More details about E* and its products are given in Chapter 7.

4.7 Analysing the data

The fieldwork produced two data sets which needed to be analysed both independently and together. Once each data set had been analysed, the two could be joined together to present a rich picture of some of the activities of agile software engineers.

The ethnographic data was, primarily, in the form of contemporaneous field notes. There were some photographs and a small quantity of notes made after the day's fieldwork was complete. These notes were to provide useful context and background and a structure within which transcribed conversations could be understood. The process of moving from field notes to writing ethnography is, in the words of Crang and Cook (2007), "an informal process of piecing things together, figuring things out". Working from the data in this way is, Crang and Cook writes, "a creative, active, making process".

The analysis of data such as field notes should be a process of discovery for any analyst. As a domain expert I was aware of the danger of missing things which were "obvious" to me, yet which might be analytically important had I noticed them. Sharp et al. (2000) are also software engineers who have used ethnography and discourse analysis and who encountered the same problem. They presented data from one of their studies *and* their analysis to a discourse analysis group who uncovered different phenomena and who created different interpretations to those of Sharp et al.. I was working in a multidisciplinary team with supervisors who are social scientists and avowed out-

siders to the world of software engineering. They were able to provide an outsider perspective on the data and analysis and to point to areas which needed more work to reveal the subtleties of the engineers' practices.

From all of the data I wanted to find talk which formulated action, grounded action in the participants reality or which explained or accounted for some aspect of their development practice. I had many hours of recordings to analyse and at the start of the process I didn't know which interactions would be the most analytically useful. I began working with the recordings by listening to them all to find sections which interested me. My field notes were examined to see if those interactions had interested me at the time. Those moments which had both produced interesting recordings and detailed field notes were selected for detailed analysis.

The detailed annotated transcriptions were analysed through a close reading which focused on analytically interesting moments. Those moments were selected because they caught my interest in the field or because they related to other moments which I had previously found to be interesting. In some cases the interactions which were selected for analysis were taken from across cases if there appeared, initially, to be similarity between them. Other interactions were taken from within a single case study but separated by some days or involving different participants, again on the basis that they appeared to contain similar phenomena.

The phenomena which were interesting were those which are brought to the fore by ethnomethodology and which relate to the primary research questions in this work. When developers share their understanding of their work the things which they say are indexical of the participants' understanding of

their work. Interactions in stand-ups and other meetings show how developers are accountable to the meeting and for their work. Analysis of the talk-in-interaction within meetings can be used, for example, to reveal power structures within the companies, interpersonal relationships, personal identity and more. The social order of an organisation is constructed through its members' talk, a directed analysis of that talk can be used to explain that social order. Activities such as the coordination of work necessarily depend on the social order of the team which is doing the work. This ethnomethodological perspective was used to sample the transcriptions to find analytically interesting passages which had not been highlighted during the fieldwork or in previous sampling.

In some cases I saw the developers came back to similar work or ideas over a number of days or weeks. Where the analytically interesting moments were linked to other moments the set of linked items would be uncovered and transcribed. The process of repeatedly winnowing the recordings yielded a *theoretical sample* of the data, (Ten Have; 2007), in which an analysis could be grounded, (Glaser and Strauss; 1967).

A comprehensive data treatment such as this gathers as much data as possible, building a *corpus* of which just a fraction is used to generate a provisional analysis. The analysis is compared to other data within the corpus and modified as necessary. Such a comprehensive treatment is well suited to the discovery of phenomena within relatively structured situations such as the development of software.

Using multiple data sets provides greater opportunities for theorising and can provide validation through triangulation and supporting examples. When

less data or fewer sources are available or analysed work can be criticised for limited coverage or for a lack of rigour. I was able to integrate the conversation analysis with wider ethnographic materials from the fieldwork. The ethnography was rich with organisational and cultural detail which provides details of both the immediate context within which the talk is being produced and of the wider organisation. The transcriptions were also richly detailed but with the minutiae of the talk. The important piece of the analysis is the integration of the two data sets.

4.8 A mixed methods approach

The methodological approach which is outlined here uses a variety of methods which are taken from across ethnography, discourse analysis and converdation analysis. Each of the approaches comes from its own epistemological tradition and is designed to answer questions from within that tradition. Can taking an eclectic approach which mixes qualitative methods be justified here?

Research in Critical Discourse Analysis, CDA, has established mixed methods as an academically justifiable approach. Indeed, mixing methods is at the heart of CDA because its critical focus requires detailed conversation analysis alongside "a theorization and description of both the social processes and structures which give rise to the production of a text", (Wodak; 2001). Fairclough (2001b) expands the possibilities which mixed methods engender to suggest that "co-engagements on particular aspects of the social process may give rise to theory and method which shift the boundaries between different theories and methods".

Mixed methods provide access to talk-in-interaction through conversation

analysis and to the context of the production of that talk. Mason (2006) writes that context is important in analysis because it

> means associated surroundings and the concept of "association" is crucial here. The analysis needs to be able to show how these elements are connected to the issues and concerns of the study, and hence in what way they are contextual rather than coincidental. Mixed methods can help us to explore these relationships, and being able to do that can significantly increase the power of our explanations.

Denscombe (2008) suggests that mixed methods "produce a more complete picture by combining information from complementary kinds of data or sources" which, in turn, lets the analysis develop through the use of "contrasting data". In the view of Esbjorn-Hargens (2006), mixed methods are "an expansive and creative form of research, not a limiting form of research".

Quantitative research in the positivist tradition is concerned with validity, reliability and generalization, (Tobin and Begley; 2004). Its findings must not only true, they must be shown to be true and to be based on analysis of the data which are presented. Such findings can then be generalized to other situations. The rigour of the scientific method is sometimes rejected by researchers who work in a qualitative way as part of a rejection of positivism. Without rigour, research cannot demonstrate integrity and competence and could be seen as little more than journalism.

A rejection of validity and reliability opens qualitative research to the charge that it is not science. Some researchers have adopted the notion of "goodness as a means of locating situatedness, trustworthiness and authenticity", (Tobin

and Begley; 2004). Mixed methods research allows for triangulation between data sets and for more rigorous research because findings arise from more than one source.

Although CDA's concerns, which Wodak (2001) identifies as power, history and ideology, were not the concerns of this research, its mixed methods approach to analyzing social relationships provided strong justification for using a similar approach. The research questions which inform this study were questions of "becoming" agile, of "doing" agile and of "being" agile. Moreira (2013) describes the process of introducing agile into an organisation as being a project in its own right, a project which leads to transformation across the organisation. A team which uses agile techniques as working practices without changing its organisational culture is "doing" agile. To truly "be" agile, Moreira suggests, "organisational values and individual behaviors need to change".

The case studies examined the transition to agile, the use of agile to coordinate work and the lived experience of programmers using the nominally agile techniques of pair-programming and daily stand-up meetings. Through the integration of data and analytical methods from different areas of social science the realities of becoming, doing and being in agile projectscould be revealed, producing richer results than would arise from the use of a single technique.

4.9 Summary

The practical and professional activities of software engineers can be studied in many different ways. A growing body of researchers is using the qualitative

approaches developed by social scientists to develop an empirical understanding of those activities. This approach presents a challenge to positivist, qualitative approaches which are more typical of software engineering research.

This research used empirical methods to uncover what it is that software engineers *do* by examining their sense-making about, and coordination of, their work as they engage in it. Data would be gathered through field observations and audio recordings which would then be integrated into rich descriptions of practice.

Three different case studies were selected for the research programme. The largest company, Z*, was moving from a waterfall and project management driven approach towards using Scrum. This company was studied before and during the transition to reveal how patterns of work and communication which were previously problematic could be transformed through Scrum.

The second case study was of a mature Scrum practice at A*.com. In this case a daily stand-up meeting would be recorded and subjected to a detailed conversation analysis to show both how the team members talk about work and the work which their talk does for them.

In the final case study, a software house which is agile but not bound to a particular method, is examined. This company, E*, uses pair programming as part of a pick-and-mix selection of agile tools. One pair of developers is followed over a number of days as they understand, test and modify some legacy code.

Detailed discussion of each of the case studies follows across the next three Chapters.

Z*: making the agile move 5

5.1 Introduction

Agile methods represent both a practical and a cultural challenge to conventional notions of how the work of software developers should be structured and managed. Agile approaches can change the power dynamics within an organisation because some of the control over daily work moves from project managers to individual programmers or to small cohesive teams. Relationships with customers or end-users may change as they become more tightly integrated into the development process and as their more pressing needs receive attention ahead of longer-term but lower priority requirements.

Making the agile move is conceptually simple and culturally appealing for many programming teams. For others, that same agile move, and everything with which it is associated, is anathema. Even when teams move to agile they seem to maintain a structured top-down wrapper around their agility. The existing studies indicate that agile methods provide a smörgåsbord of tools, techniques and approaches from which they cherry pick those which appeal.

Both academic researchers and practitioners of agile methods remain interested primarily in the practical detail of their implementation: which method,

or methods, might a team choose, what changes need to be made to their processes and what benefits might they see. The authors of agile methods have written books and papers explaining the rationale behind their approach and the benefits it brings, alongside details of the practices and project structures the method defines. For example, Schwaber (1995), Beck (2000), Sutherland and Schwaber (2007) on Scrum and XP act as both tutorial and polemic. Authors such as Martin (2003) identify best-practices for teams who are moving to agile methods. Others, including Cohn and Ford (2003), Benefield (2008) and many more, review what happens in particular organisations after the agile move.

These studies and tutorials tend to leave a gap in their discussions of agile. They describe the types of situation, for example changing requirements, which lead to the though that agile development might be a good idea. Some of these authors also describe the team once it becomes agile. What is lacking is an examination of the period during which the team changes from a traditional approach such as waterfall to an agile approach. The changes which agile methods bring are not just matters of engineering, they bring different thinking about process, about produce and about the roles and requirements of users within the project. These changes combined suggest that agile development might bring with it a different culture to that of other ways of organising the work of software developers. These changes are themselves matters which are worthy of study.

This Chapter follows a small software house as they introduce Scrum. It will look at the company before they made their agile move and once they had done so. The changes in working practices, communication patterns, relation-

ships and organisational culture which accompany the use of an agile method will be revealed. The Chapter includes the views of key informants such as senior developers, project managers, programmers and testers and looks at key the Scrum practices of stand-ups and reviews.

5.2 The company

Z* is a small software house which make number of products but their main one, at the time of the fieldwork, was DVD authoring software. This software was sold to film and television companies across the world although their main market was in Hollywood.

Z* employed a relatively small staff. Three groups mattered during these observations. Ten programmers wrote code for a variety of products, a similar number of QA staff tested applications before release and three people, the product managers, liaised between customers and developers. The roles and relationships within the staff are described in Section 5.6. Towards the end of the field work the company made a number of significant financial and staffing changes which had a dramatic effect on their personnel and structure. These changes were a significant factor in ending my engagement with Z* on this work.

5.2.1 DVD authoring software

Major film studios and television companies are the largest producers of DVDs. Each of them releases hundreds of titles every year ranging from individual movies through to box-sets of television series. Successful titles can be re-released with new material added or with new trailers pre-pended. When the

material on the disc changes a re-release isn't a simple process, the disc has to be redesigned and thoroughly tested so that, for example, the correct language tracks are added.

A DVD is built from *assets* such as sound tracks, video, subtitles in several languages, menu buttons and pieces of text. These are brought together in a project which details how they are structured and how the viewer interacts with them along a time-line. Creating a DVD, usually called *authoring*, is a specialised process which Sennett (2008) would identify as a type of craftwork. Few people can do this well – probably a few thousand world-wide – and demand for their services is very high. The DVD market is a major revenue earner for film studios but those earnings are not spread evenly throughout the year. The majority of DVD sales occur in the weeks preceding Xmas. The studios rush to have their Christmas range of titles available on time but may not be able to produce the requisite source materials and assets until shortly before the release date.

Z* realised that much of the authoring work can be mechanised and that good tool-support might mean that authoring is de-skilled and becomes both faster and cheaper. This de-skilling process has been a normal part of industrialisation, (Rolfe; 1986, Barley; 1988). By simplifying complex activities IT solutions should increase efficiency which has been touted as one of the major benefits of IT, (Brynjolfsson; 1993). Increased efficiency is available in DVD production if the process can become more like word processing and less like design. For DVD publishers the seasonal sales peak at Xmas presents something of a headache: they must get their titles authored early enough so that disks can be pressed and distributed to stores but sufficiently late in the year to

include summer blockbusters. The laborious manual creation of DVDs does not lend itself to high volumes or tight deadlines. Consequently there is demand from studios, if not from DVD authors, for a tool which can simplify the process.

Z*'s main product was a browser-based application built using the Google Web Toolkit, (GWT). GWT applications are written in Java which is compiled into a mixture of HTML controls and JavaScript. The HTML controls are displayed as a Web page with the JavaScript interpreted by the browser. In theory GWT applications have advantages of platform independence and portability over other types of dynamic HTML. With the creation of JavaScript libraries such as JQuery, Angular.js and Script.aculo.us the creation of interactive Web applications based on AJAX/HTTP and HTML has been greatly simplified. And developers who want to avoid writing JavaScript themselves can now use intermediate languages such as Dart or Coffeescript which are compiled into JavaScript.

Most of these JavaScript libraries and pre-processors have little or no tool support. The great advantage of GWT for a developer is that they can code using Eclipse as if they were writing Java. The familiar structure and the support which Eclipse has for intellisense, inline syntax checking and its integration with testing and debugging tools greatly reduces programmer effort. The reduction in effort is insignificant for a single person working on a simple page but a vital saving when a team is building complex applications.

5.3 Measuring success

At the start of the project we agreed that I would help the developers and management assess the effect of introducing Scrum. The team at Z* wanted a lightweight evaluation process rather than one which was based on the collection of large volumes of data. They were happy to use their existing metrics for code quality, productivity etc. and believed that they would be able to compare the new agile team with existing teams without too much additional overhead.

The work of programmers can be evaluated using metrics such as the amount of code written per person per day or the number of check-ins to a source-code control system. Ilieva et al. (2004) compared agile and traditional projects using measures such as productivity, defect rates and even deviation from schedule. Scrum projects are often evaluated by counting the number of story points which are completed in each iteration, (Deemer et al.; 2010), or through *burn-down charts*, (Buglione and Abran; 2007). Individual metrics can be used to assess documentation, meetings, communication with customers, QA processes or the creation of designs but there isn't a straightforward way to amalgamate this range of measures to give a holistic view of a project.

There are many less quantitative approaches to evaluating work, (Cohn and Ford; 2003, Coram and Bohner; 2005). Sometimes qualitative answers arise from empirical work. Muller and Padberg (2004) investigated why it is that programmers working in pairs seem to be more productive than those who work on their own and arrived at the qualitative answer that it is due to "feelgood".

The management at Z* weren't looking for a quantitative assessment of

their changes – they were already gathering numerical data to help them understand their development processes. Instead they wanted to use a light-touch approach which would give qualitative results so that they could understand if, and how, their organisational culture was altered when they made the agile move. They wanted to look for phenomena such as "feelgood" and job satisfaction.

A broadly ethnographic approach was chosen to reveal the social and power structures at Z*; the ways in which people ordered their working lives; and relationships within the company and with its clients. Changing working practices will be reflected in changing interactions and patterns of talk. Ethnomethodology shows us that interactions and relationships are built and maintained through situated actions and embodied in talk. They can be revealed through analysis which is informed by ideas from the discipline of Conversation Analysis.

This approach is interesting because the focus is on evaluating the staff's experience of agile rather than evaluating productivity gains or improvements to the quality of the code. The management were, as will become clear, reasonably happy with the amount and quality of code being written. They were introducing agile methods precisely because they wanted to effect a cultural change.

5.4 Negotiating access

Before we could work together the team at Z* had to come to trust me. They needed to know that the fieldwork and later analysis would be fair and accurate and that there commercial sensitivities would be respected.

In December 2007 I went to a meeting in the boardroom at Z* with two of their senior developers, Rob and John, alongside their Head of Human Resources, Hilary. I had previously met Rob and John when we were trying to find some projects on which we could collaborate. We met only briefly that time without establishing a meaningful relationship.

When Rob, John and I first met we spent some time talking about software development, programming languages and project management. Although our conversation was relevant to our proposed collaboration it wasn't necessary, it wasn't part of the collaboration. We were each ensuring that each of us was "sound", that we knew what we were talking about. People meeting each other or working together have an interest in the ways in which others perceive or evaluate them. We use a range of impression management strategies to try to control those perceptions and evaluations, Leary and Kowalski (1990). When we get a favourable response, or one which meets our expectations, from others our impression management approach is reinforced, Gardner and Martinko (1988).

When we meet people through our work we are, naturally, interested that they see us as somehow skilled or professional and consequently we work to give that impression. They will usually be doing the same. When each party creates the right impression they will be more likely to trust each other as happened as Rob, John and I talked. We were able to begin to trust each other once we discovered shared ideas about programming and about the use of agile methods and I was able to present myself as an insider with legitimate claims to understand their work. If we hadn't built that initial trust we might still have worked together but with less confidence.

During the meeting John and I did most of the talking. Hilary was mainly silent in this meeting although she was friendly enough. She wanted to see the "happiness" of the developers increase as she felt this would increase both quality and productivity. The most valuable resource in an IT business is the technical staff and treating developers better makes better developers, (DeMarco and Lister; 1999). Hilary and John had both read and been strongly influenced by *Peopleware*. In particular, they felt that the environment within which programmers work should be conducive to that work. Hilary talked about making the structure of the office supportive of both individual and team working. John was more interested in managing schedules and workloads so that the programmers could produce their best work.

The team from Z* started by outlining the way they currently operated. Developers were split into three groups. One built tools, applications and the GWT-based templates which are used to author the DVDs. A second group provided quality assurance, QA, in the form of testing and evaluation. The third, much smaller group, provided support to customers. The relationship with the customers was managed by three Product Managers in Sheffield and a sales team in California.

Each developer worked concurrently on products for a number of different clients. Z*'s application was not a generic authoring tool. Because of the complexity of the menu systems and content on DVDs the tool had to be customised for each client and for each application. Spending much of their time customising the tool meant that the work was largely reactive. Studios would require fixes or changes immediately. As a small supplier into a niche market Z* had to respond straight away. Planning work is always difficult in and

environment such as this but at Z* it was felt that the problem was exacerbated because developers were not structured as teams. John, in particular, believed that working in properly formulated teams would give cohesiveness and improve efficiency and performance.

In theory the customers were only to liaise with Z* through the Product Managers but in reality they often talked directly to QA or to development. The drawing on the left of Figure 5.1 shows some of these communication pathways. These communications were described by John as too often ad-hoc and undocumented which caused unnecessary duplication of tasks and requirements.

During our discussions we concentrated on culture and working practices. We spent very little time talking about productivity or quality. I was told repeatedly that they had a problem with internal conflicts between departments and disagreements within projects. John said that these were the "driver" for the introduction of Scrum. In particular there were problems "between sales and development" but during this introductory meeting no-one would say exactly what the problems were – they were described only in the vaguest of terms. The Z* team said that they wanted an me to discover the problems for myself and to bring an outsider's perspective on why they were happening.

Sutherland and Schwaber (2007) write that some Scrum teams achieve a state of "quality without a name". This phrase, borrowed from Alexander (1979), describes that quality of a design, object or process which is sought after but which cannot be named. In successful Scrums it "can only be spoken of as a set of core values – openness, focus, commitment, courage, and respect", (Sutherland and Schwaber; 2007). Some of these are very similar to

the core values of Extreme Programming discussed in Section 2.5.1. By bringing a member of the sales team into a team of developers John hoped to break down some of the barriers which had grown within the company.

I agreed a time to turn up to start observing, was given a security fob for access to the offices and took a non-disclosure agreement to take away and sign. I was to go in one morning each week throughout the initial couple of Scrum iterations, more often if interesting things were happening. This schedule was agreed because it fitted nicely with our respective workloads. I was busy, Z* weren't sure exactly how the Scrum would fit into their other work. The developers who worked in the Scrum would also be working on other projects and because it was their first attempt they were likely to start off relatively slowly. Being continually present at their office probably wouldn't yield better data than I would get from well targeted visits.

5.5 The first day

On the first day I got to Z*'s office at 10, early enough for the day to still be starting, Z* run a flexi-time system of working hours, yet late enough that most people had read emails, caught up on over-night developments and were settling down to the new day's work. John had asked that I spend a few minutes with him and that we do that once he had checked his messages.

I spent the first forty minutes being shown around by John, introduced to the building and its inhabitants. The layout of the building was interesting and would become an important factor behind some of the problems which the introduction of Scrum was meant to solve. When I was first invited in, the Z* team felt that the structure of their working space was inhibiting what

they were able to do and that it promoted an unnecessary amount of internal conflict. They had recently moved into this building from a larger one a couple of hundred metres away. Team locations had been made concrete quite soon after the move and people were reluctant to move their desks around without good reason.

Z* occupied the upper four floors of a nine story 1980's tower block.

- Level six housed the reception desk, administrative functions and the board room.
- Level seven was software development, quality assurance, technical support and the technical author.
- Level eight was the product floor with the product managers, product QA and product development. It also housed a room which contained all imaginable DVD players from around the world which were used to test disks before templates are sent to the studios.
- Level nine was the server room, a large meeting room which was said to be used only for pep talks from the Managing Director, and a small meeting room which became the focus of Scrum activities.

Most of my time was to be spent on level seven or in the smaller meeting room which was to be used for daily stand-up meetings. The upper floors of this building form a single doughnut shaped room which wraps around a central core which, in turn, holds the lifts, stairs and an entrance vestibule. Twenty five people work on level seven although it is large enough to hold half as many again and still provide each of them with plenty of space. People work in clusters of four desks regardless of their role. The desks are large enough

to accommodate a dual-monitor set-up as standard with space for note taking, text books etc. The cliched partition-formed cubicles, which were railed against in DeMarco and Lister (1999), are not used here which makes this large room feel open, light and airy.

The other levels have even more empty space. When I talked to John about this he said there "really isn't room to move everyone around". They won't be co-locating the Scrum team. Partly this is because the members of the Scrum will also be working on other projects

The first thing that struck me on entering level seven was just how little noise there was. The space was large enough to swallow the whirr of PC cooling fans and the air-conditioning was not roaring away. One of the noticeable features of the silence was that the phones never ring. I found out later that internally people communicate using instant messaging and that communication with customers in Hollywood usually happened over email or in conference calls at the end of the UK working day.

Most people sat in quiet concentration, wearing headphones to listen to their own music. When I talked to individual developers it became clear that the problems they were solving were relatively trivial when compared, for example, to systems-level programming. The developers said that they had neither the desire nor the need to write lots of code each day and, consequently, they worked at a relaxed pace.

Quiet conversations bubbled up then fade away. They were usually about work but sometimes about external matters such as, on this day, my presence, pension schemes (an adviser was coming in) and lunch. The work conversations were usually about internal processes: "Did you check it out of CVS?",

"Are you using the emulator?" rather than about design or algorithms.

Everything was inward-looking and focused on the tasks at hand. The blinds were drawn even though there is a good view over the city from most windows. There were no notice boards and only one bookshelf of manuals. This lack of clutter might have been because they only recently moved into this building and it hadn't yet been personalised but John told me that it was not unusual for their offices.

The staff assumed different sorts of role at different times. I saw a conversation between two developers who were debugging some code. The conversation was about how the intricacies of how DVDs work and how this affected their code. One of them was advising the other, but a few minutes later he was asking for help from a colleague with some code of his own. They discussed how the product handles faulty data, a situation they cannot resolve at this point. As they bounced ideas off each other they began to formulate a new direction for future work. Developers helped each other in episodic and *ad hoc* conversations throughout the day.

The calm atmosphere was only broken when one of the product managers burst in to the office. He carried an atmosphere of aggression and conflict with him which had a marked effect on the room. He rushed up to one of the developers demanding that he send some information to a customer. When told that it had already been provided he was very forceful in saying that it "needs" doing again. He then made a fuss about the developer being available for an instant messenger chat with the customer at 5 o'clock. After a couple of minutes he rushed off and the atmosphere quickly returned to calm. The programmer appeared unaffected by the interruption, he returned to his work

almost as if it had never happened.

This incident was a classic example of the communication problems within the company. Both men clearly thought that individually they had done what was required. The developer had talked to the client, the product manager had talked to the client and to the developer. Although each of them was talking neither of them listened: there had been a small communication problem of the sort which happens all of the time and which is usually easily corrected with a phone call or an email. At Z* it blew up into an argument.

The product managers worked on a different floor to the developers. They saw themselves as buffering requests from clients and as simplifying the information flows into development. Section 5.6.4 is an examination of the role of the project managers.

5.6 The staff

When I first met them, Z* was a reasonably successful company. They were shipping products and had a small but growing customer base. The customers were prepared to provide feedback into Z*'s development processes and to work with them to improve their products. I had expected to find a certain amount of chaos with product not shipped or shipped incomplete and unusable. I thought I'd see a *Death March*, (Yourdon; 2003), since these are said to be very common in the multimedia world. In a death march requirements, usually scheduling but sometimes functionality, cannot be met in the time available. The staff working on the project know this but make futile attempts to achieve the deadline. Often towards the deadline extra staff are brought on to the project team but Brooks (1995) demonstrates that it rarely helps.

Instead of death marches at Z* I saw projects which ran up to the wire, sometimes requiring extended deadlines, but which were being delivered without undue panic.

Often agile methods are introduced out of adversity whether that is fear or failure, (Schwaber; 1995). Scrum is fast and cyclical which means that, of all the agile methods, it is the one which is best suited to struggling projects. When Scrum is used alongside techniques such as test-driven development and supported with high-quality modern software management tools teams can gain the velocity they previously lacked, (Sutherland et al.; 2006).

Z* didn't have failing projects, they were delivering product to their customers in a timely fashion but they did have communication problems. John's answer was to reorganise the way that they worked through Scrum. Alongside this changes were made to the way in which they communicated with their customers. Communication had followed complex patterns. in which customers talked directly to a range of staff including individual developers. This led to duplication of effort, mixed messages and confusion. A more formal process was being put in place in which customers spoke only to specific representatives of Z* and they, in turn would communicate with the appropriate engineers. Figure 5.1 shows the old pattern on the left, and the new simpler structure on the right.

5.6.1 Project management

Software projects are different to other engineering endeavours because requirements change repeatedly during most projects and because software is both complex and intangible, (Jurison; 1999). Project management is generally

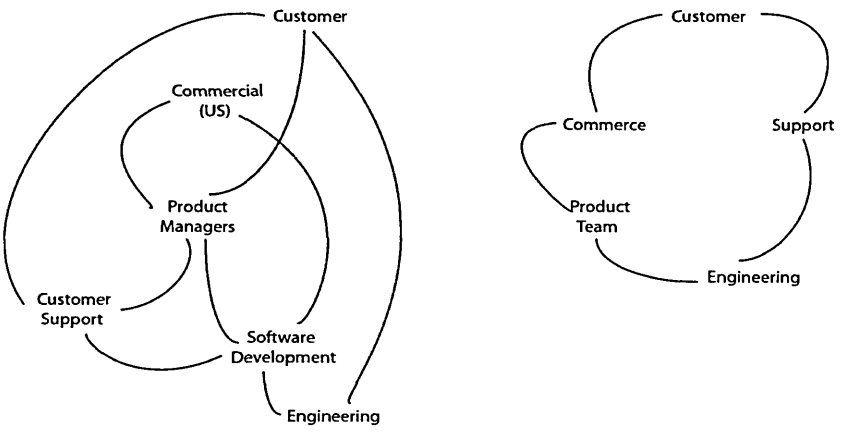


Figure 5.1: Old and new communication structures

seen as a key part of any software development process, (Software Engineering Institute; 2010). The project manager's role is "the planning, organizing, directing, and controlling of company resources for a relatively short-term objective that has been established to complete specific goals and objectives", (Jurison; 1999). Broadly, project managers use a variety of approaches to try to ensure that the client receives the software they expect on time and to cost. The project manager is there to understand and manage all of the factors which impact upon the team and on their ability to "get the job done", (Jurison; 1999).

At Z* they had an experienced project manager, Dave, to oversee the operational aspects of product development. These included coding, testing and QA, and technical authoring. Dave saw his role as ensuring that resources, primarily staff, were used effectively so that product could be delivered on time.

When high priority, short term contracts arrived there was a sense of panic because all of their developers were already allocated to other tasks. Kraut and Streeter (1995) wrote that one of the major problems facing the software industry "is the problem of coordinating activities while developing large software

systems". Systems don't have to be large to be complex and they don't have to be large to be difficult to manage. In a company like Z* co-ordination problems can arise when managing large number of relatively simple projects with limited technical resources.

Dave explained to me that people were given tasks to achieve and were expected to get on and get them done, managing their own workload to do so. Each task had an estimated duration, time to completion, associated with it and, often, a deadline. People would put in the effort to get work done without being forced to do so, even working extra hours if necessary. The typical developer spends around five and a half hours coding, testing and so on each day with the remaining two hours taken up in meetings, emailing and talking to colleagues. This amount of work is, in Dave's experience, "typical across the industry". The difference at Z* was, he claimed, the amount of personal responsibility developers take to manage their work. In fact, the split between engineering and other activities may be even more typical than Dave thought. Perlow (1999) found that the developers she studied spent around two-thirds of their time on engineering, typically working alone, and the remainder on a range of interactive activities including email, phone calls and meetings.

The workload in *engineering* was driven by business cases, without support from *the business* products were not built. The creation of a business case was a process of negotiation between the sales team, product managers and Dave. Each project had an entry in a Wiki with the rationale, key tasks and business case attached and viewable by all staff.

Dave's approach to project management was to focus on the resources. Project management methodologies have large sections devoted to resource

management because if you don't know what resources you have available you cannot know how to deploy them to fulfil tasks and projects.

All of the raw data about projects was held in either Microsoft Project or in the bug tracker. These two applications didn't give a unified view of the resources (primarily the people) so Dave had written some scripts which imported data from them into an Excel spreadsheet. Each task in a project, including bug fixes, was assigned to a "Tech. Lead" who assigned them to the appropriate developer. A "Time To Complete" was estimated although it was never clear who had the final decision on these estimates. The estimates were fine-grained, sized in thirty minute blocks.

Using his spreadsheet Dave could see what everyone was supposed to be doing throughout the day and, for some things, up to six months ahead. He tracked holidays, sick leave and so on using the same systems because peaks in those affected developers' availability and hence the amount of work that could be handled. Dave showed me that annual leave peaked at Christmas and in August and that there was a lot of illness in November. All of this information was publicly available on a Wiki so that everyone could see their colleagues' workloads. The types of "administrative coordination" done by project managers affects the performance of a team, (Faraj and Sproull; 2000). But often administration's benefits can arise without the support of complex tools. "[S]ubjective assessments of effectiveness provided by knowledgeable managers have a high level of convergence with other objective measures of performance", (Faraj and Sproull; 2000). Dave's common-sense would have told him that requests for leave peak at Christmas and during the school holidays, for example.

When I first met him Dave was involved in planning the workload of the technical author. They were looking at Gantt charts in Microsoft Project and trying to plan a workload over the next few months. This was to prove to be a useful example of how projects were run at Z* – and one that some developers were desperate to change. Although work came in bursts and new contracts could arrive at any minute there was a desire within the management to know what everyone was going to be doing for several weeks or months ahead. For some of the managers, including Dave the Gantt charts held a canonical truth which couldn't be questioned. However some of the developers said that Dave over managed and had an obsession with his Gantt charts and that this didn't really help anyone. Certainly, when Dave talked me through his work it was all about the data and the software. I didn't notice any comments about people, projects or products.

In tightly planned organisations the project manager can become a single point of failure. Dave was sure that wasn't the case at Z* because there was so much software support and because the data was so open. Others were less confident and told me that they wanted to become less reliant on Dave's Gantt charts and spreadsheets. Some of the developers, in particular John, felt that too much reliance on data, data which could be "gamed" because it was public, could restrict how people worked. Al-Zoabi (2008) writes that "too much strictness kills creativity and initiative spirit in the project team". However a lack of data can lead to a lack of control. Software engineering often follows other engineering disciplines where projects still fail but "[c]urrent thinking in the project management community is highlighting governance issues and poor and inadequate risk management", (Lawrence and Scanlan;

2007).

Dave felt that in the move to Scrum he would be affected more than most people. He was clearly concerned about the impact of Scrum, expressing his view that there would be less data and, hence, less control of projects and, especially, over risk.

The introduction of an agile method isn't always a neutral process. Some groups in the organisation benefit whilst others may find their role or status is threatened. Benefield (2008) saw one Scrum team which was so zealous in rejecting external influence that the company management had to restructure the project to regain some control. The managers didn't look at the success, or failure, of that Scrum through traditional measures of quality or productivity. They simply saw that control was moving away and acted to reassert themselves.

At Z* significant power over the daily management of their own work moved to the Scrum team and away from Dave. Perhaps unsurprisingly Dave's were the loudest objections but despite them the change had sufficient backing from senior developers John and Rob, and from Hilary, on behalf of HR, that the Board of Directors agreed to a trial.

5.6.2 The developer

On one visit I spent time with Ben, one of the programmers. He had worked for Z* for three months at that time and talked me through the company's existing approach to software development.

The programmers used some standard tools for source code management and bug tracking but were free to use any tools they chose for coding. At this

time Ben was using Eclipse to write Java and WingIDE to write some Python. He didn't like using WingIDE which made me wonder why he didn't install some Eclipse plug-ins so that he could code both Java and Python in one application. IDEs often enhance a programmer's productivity and simplify coding by, for example, supporting *intellisense* which suggests possible classes and methods as you code, providing context-sensitive help and having integrated debugging. A proficient programmer will find a set of tools in which they are competent and will work to master their use. If Ben had used Eclipse for all of his work he would have become proficient in it and become a more efficient programmer, (Hoover and Oshineye; 2010).

Later in this session Ben would switch to a plain text editor called Notepad++ to write some XML even though Eclipse has excellent support for XML. Ultimately it may not matter that someone doesn't fully use their tools but a developer who is competent in their programming environment will do less work and can concentrate on the problem domain and on their code rather than on the tool. Z*'s developers seemed to struggle to make the move from apprentice to craftsman, (McBreen; 2001, Sennett; 2008, Hoover and Oshineye; 2010). That moves requires support and motivation, Weinberg (1998), but increasing ability leads eventually to the production of higher quality outputs, (Sennett; 2008).

Ben was writing a template for a client's DVD using GWT. He told me that Z* had switched to GWT about six months before after using raw JavaScript for a number of years. He had been with the company for just three months and this was his first exposure to both GWT and to DVD authoring. Before working here he had been a Web developer at a local ISP. Software engineers tend

to be highly mobile with careers which take them through a number of technologies and through work on many different types of application, Ó'Riain (2008). Moving from writing Web code at an ISP to crafting DVD templates is a normal type of career move.

GWT itself is well documented, there are some good books and the forums at Google are very helpful. But the templates written at Z*, the actual code on which the developers work, contain no comments, have no supporting documentation. There isn't even any design documentation: Ben said "I haven't seen or heard of any" when asked about design. The code is so complex, and the DVD authoring domain so esoteric, that John, the lead developer, told me it takes new recruits *at least* three months to get up to speed even if they are experienced programmers. When Ronkko (2007) write about searching for indexicality they are defining the activity with which Z*'s developers engage. To understand the design of the product they must understand the code but that code is both complex and esoteric. Developing a facility with that code requires that significant time and effort are invested. The staff accept this long period of learning as a normal part of their culture. They don't see any problems with the associated costs either financially or in wasted effort and lost time.

Documentation provides a self-administered from of control. Programmers "must have a common view of what the software they are constructing should do, how it should be organized, and how it should fit with other software systems already in place or undergoing parallel development", (Kraut and Streeter; 1995). This shared view comes either from formal design documents or from comments written into the source code or, in theory, from both.

However documentation is often not kept up-to-date so that when the code or design changes the documentation lags behind. Design documentation is not just held in formal structure, but also in informal notes, in the minutes of meetings and so on. These informal documents are not typically indexed and can't easily be used by someone who is unfamiliar with them. This forces people to talk to each other about the context within which the documents were written and about what they mean, (Hertzum and Pejtersen; 2000). Agile methods introduce many additional informal documents such as index cards, post-it notes or story cards. These are often used transiently, being placed on a Kanban board for a few days then removed and thrown away once "completed".

Chapter 7 follows two developers working as a pair as they rework existing code. The Chapter will demonstrate that talking about a section of code can reveal both the content of the code and the context of its production in ways which may be richer than formal documentation could provide.

Many agile methods "eliminate unnecessary activity devoted to documentation" and instead rely "on oral communication [which] emphasises the importance of developers' individual memories", (Sharp and Robinson; 2004). A traditionalist might baulk at this but Kraut and Streeter (1995) argue that talking and the creation of strong interpersonal networks and lots of informal communication correlate very positively with better project outcomes. However, where there is minimal documentation there is also minimal long-term knowledge. Because developers move between companies or between roles within the company so frequently their specialist knowledge is easily lost. Rus and Lindvall (2002) write that "[t]he major problem with intellectual capital is that it has legs and walks home every day. At the same rate experience walks

out the door, inexperience walks in the door".

5.6.3 Quality auditing

The QA manager, Chris, had two roles: he managed the testers and he managed the support staff who work with customers. Because he started out as a tester himself he felt that he had good understanding of what they needed to get their jobs done. Less formally he saw his role as "insulating the [people in QA] from problems". The QA role was one in which tasks were never immersive and there was always something else to be done. In fact, Chris said "I work in Outlook" and showed me how he spends his time in the Calendar application juggling "[l]ots of meetings".

Chris talked me through the process of implementing a template. The customer supplied Z* with a rough outline of their requirements including the number of episodes; subtitle languages; trailers and other material which they wanted to include; and an idea of the menu structure for the DVD. A formal statement of requirements was written collaboratively by the sales team, product manager and customer. The requirements were formed into a template, using GWT, by the developers. Testing then began. The tests were specified early in the process and taken directly from the requirements document. QA spent their time using the template, making sure that elements such as menus or subtitles work correctly. Once the template met the specified requirements it was shipped to the customer who added the content and built the DVD. Chris said "instead of $10,000 they can do it themselves in half an hour". The templates could be re-used so that a single template was suitable for an entire box set, the only change across the disks being their unique content.

The QA testing at Z* was analogous to integration testing in conventional software environments but here it could, in theory, begin earlier in the process. It tended not to do so because development often took longer than expected. In a typical eight week project the last two weeks were scheduled for QA but that time was often eaten up by development. This led to rushed QA and meant that there often wasn't enough time to work on properly fixing the problems which QA threw up. When QA found a bug they wrote a description of it, including how to reproduce it, in the bug tracking software, assigned it back to a developer and gave it a priority. If the problem was found shortly before release the QA manager had to negotiate with the developer and the product manager to decide if, how and when the problem was going to be resolved. Chris said that "QA want to ship quality product" but the product manager gets the final say. The vast majority of staff will want to ship quality products but QA is the only part of the company which explicitly "has that as its goal".

The product specification was written in plain English. I was told that specifications were "often" vague or ambiguous although no one had hard numbers on precisely what "often" meant here. From the testing department's point of view the specification needed to be converted into a tightly worded requirements document and a formal test plan. These were mainly written by the Product Managers and agreed to by both development and testing.

The requirements specifications usually included a flow chart which shows how the DVD could be navigated. Test cases were built on this with one statement or one part of the chart sometimes leading to a number of different tests.

The software testers managed their daily work using the bug tracking software. This software acted as a repository of test cases, scripts and planning

documents. If a tester needed to know more detail about a disk they found it here. When I watched the testers working on products it was never clear if they were driving the bug tracker or if it drove them. It was in control at least some of the time: when a problem was signed off, signed out or signed in the software emailed everyone whose name was attached to the problem. This senior tester found the software useful because it gave him an easy way of seeing exactly what he and his team had done.

The testing process was both intensive and time-consuming. The navigation through every DVD template had to be tested in all possible combinations following a test plan which was written by the product managers and customers. The single largest problem for the QA department was that their work could only be fully carried out at the end of the development process. A typical waterfall project lasted six to eight weeks but QA only started work in the last two weeks. If they had problems with the specification or the test script they could struggle to complete their work before the deadline. Since their role was, in part, to find all of the bugs in the software they were often referring templates back to the developers when there was little time to fix and test the bugs. However the testers oriented strongly towards the goal of raising quality so that only high quality products were shipped but rarely had time to complete their work to their own satisfaction.

Lack of time to complete projects, and in particular insufficient time for testing, is one of the problems which Scrum addresses. Through the integration of testing and QA functions fully with the developers, software can be tested as it is being developed, (Sutherland and Schwaber; 2007). This wasn't one of the drivers for introducing Scrum at Z* but it might be a beneficial side-

effect of that move.

5.6.4 The product managers

Z* had three product managers: Sean, Justin and Julia. Justin was the person who loudly berated a programmer on my first day with the company.

The product managers' role was to agree projects with customers, sales and development. They all said to me that these agreements were based solely on financial criteria and that the financial viability of each project was an important factor for them in their decision making. The product managers saw themselves as providing a buffer between development, sales and customers. They acted as a buffer during both the initial negotiations and later in the product life cycle when customers wanted code to be modified or maintained.

All three product managers said their approach "ensured" that development only received "useful information" which they could "actually use". When I spoke to the developers and the QA team *they* said that the specification documents they received were often vague and imprecise. This wasn't a characterisation which the product managers recognised. They were confident that their specifications reflected the needs of the customers in a format which the developers could use.

The product managers said that they try to avoid a "them and us situation" with the sales team. The communication between sales and customers seemed to work well. The product managers felt that sales were "quite good" at getting information from customers. But that information was then filtered by the Product Managers before it got to either of the development or testing departments.

Z*'s customers paid for the software on a per-use basis where each use was the creation of a DVD ready for pressing. In the established DVD authoring world, film and TV studios out-sourced work to three different companies. These charged studios on a per-hour basis to author DVDs and were, obviously, not interested in using Z*'s technology which simplified and sped up the authoring process.

All of the product managers agreed that delays in development or testing at Z* lead to difficulties in prioritising work. In particular, it was always difficult to decide between fixing errors in existing products and creating new products. Despite this, one of the product managers felt that their relationship with development was better than that with sales. He wouldn't elaborate on this when pressed, instead going quiet as if he had said too much.

On one occasion I watched Sean negotiating with lead developer Steve about using Scrum. They had a one-off project which the Board had agreed to as it completed a contractual obligation. Steve wanted to run this as a Scrum but said this would cost more than doing the work in their usual way. The additional cost came from factoring in QA from the start of the Scrum, paying for a full-time Scrum Master and bringing in some outside expertise to get them started. The expert was part of the start-up costs of learning about Scrum but the cost of using them was to be loaded onto the first such project instead of being spread across all iterations. There was a long discussion between the two men. Both said that they "wanted" to try Scrum but were "held back by Z*'s financial model".

The testers were particularly critical of the "vague" specification documents they received from the Product Managers. This single factor was the

source of a lot of the disagreements between the different departments. These disagreements provided the context within which Scrum was going to be introduced yet senior staff were finding impediments to that introduction. "Wanting" to introduce Scrum was clearly a weaker idea for them than fitting in to the financial model.

5.7 The first Scrum

After my meeting with the product managers I met with my original sponsor, John. He told me that Scrum was off the agenda for a while. Then, out of the blue, I got a call from him saying that they were starting a Scrum and that they had a consultant coming over from California to work with them for a week. The cost of this consultant was the problem with which I had seen Sean and Steve struggling. Clearly they had found a way of resolving the problem.

The consultant, Michael, had an interesting role which seemed to be to bring confidence to everyone that they could succeed. He said two interesting things: Yahoo! had used Scrum and a paper was presented at HICSS '08, Benefield (2008), which showed they had succeed through getting widespread buy-in; and that measuring success is "awkward" and no-one really has a metric for it. The latter was an interesting point coming from someone who frequently works with failing projects and who travels the world selling his knowledge.

Michael was going to help all of the members of the first Scrum to understand the the new working practices, project's structure and their new roles. He was going to run a few Scrum meetings such as the first Planning Meeting and some daily stand-ups and show the Scrum Master a few techniques which were successful elsewhere.

I attended the first Scrum Review meeting, which happens at the end of each sprint, at the end of March. Nine people attended from all aspects of the product: Julia the product manager, who acted in the role of Product Owner or customer, three developers, a similar number from QA, a Scrum Master and someone from product testing. The Scrum Master was the head of QA with whom I had spent time earlier – someone who was heavily invested in getting improved communication and higher quality so that his Department could function more effectively. He was fully integrated into the team for Scrum but wasn't involved in software development which allowed him to be a relatively neutral voice in the meetings.

This first Scrum Review started by looking at the work which had been achieved on the sprint. This contrasts with the stand-up meetings such as the one in Chapter 6 which focus on immediate issues for the day. One of the developers "drove" the software they'd written during the sprint whilst the others suggested things which they wanted to look at. They had the bug tracking application open and took other suggestions from its list of tasks. When problems appeared they were discussed in a friendly way. The team spent quite a lot of time discussing functionality with Julia asking questions from the customer's perspective. The functionality being discussed was not clearly defined in the specification document whose meaning could be open to interpretation. In these discussions they reached consensus around those interpretations and how the template matched them.

As the Review proceeded a number of bugs were identified in other applications, particularly the system which compiles a template into a DVD structure. This team had no control over those applications and could do nothing

more than log the bugs. This was a problem because those bugs were delaying the completion and release of the template. This is a common situation in software development because it is so reliant upon good tool support. Whether tools are built in-house or bought in from external suppliers the whole process necessarily becomes reliant upon them. One of the arguments in favour of Free Software is the users can fix the bugs in their tool chain rather than relying upon a supplier to do so. This assumes that the user has the skills and time to debug and patch someone else's code but commercial developers are often fire-fighting their own releases. Clearly this problem is both critical and difficult and is not one that has been solved effectively, yet.

Discussion of the template involved a discussion of low-level detail from the ISO DVD specification. This sort of *domain knowledge* is key to both design and implementation. A number of software development techniques have appeared recently which attempt to exploit domain knowledge to help developers build better software. They acknowledge that ideas such as XP's on-site customer or the work of the CSCW community were well-motivated but that they didn't give developers the information they actually required. In the Z* Scrum team some of the developers had expert knowledge of the intricacies of the DVD specification – they had even had a representative on the DVD standards committee. In this meeting they acted as tutors, making sure that the rest of the team really understood relevant details.

After the product walkthrough the team began to review the user-stories. These are natural language descriptions of the use of the product and they have to match the workings of the software. During this discussion the Product Owner said that she "won't sign off anything which development hasn't

signed off". She meant that if the developers weren't prepared to say their code worked then she wouldn't make the claim and it wouldn't be released to the customer. This seemed to me to be both co-operative (we're all in this together) and aggressive (you'd better get it right) at the same time. The application bugs were identified as the most serious ones but I didn't note anyone being tasked to get them prioritised with the applications team.

After a short break the meeting moved into a sprint retrospective. In turn everyone listed those things which went well and those which went badly during the sprint. The Scrum Master wrote each point onto a separate Post-it note and stuck them onto a whiteboard. Once the set of notes seemed to be complete the Scrum Master read them all out to the team. The team discussed the results of the sprint. Everyone agreed that the product was more solid than previous ones and that this process of review and retrospection had identified a number of serious problems with the underlying template.

The team had realised that not being co-located was becoming a problem for them. The Product QA team were on a different floor to testing and development and didn't feel that they understood how everyone else was working and that they were not being given enough time to complete their work. These are typical feelings on many projects but s Scrum retrospective gives a space in which they can be aired in front of the whole team rather than festering and becoming a source of resentment.

5.8 The Daily Scrum Meeting

By the time that I went back to Z* to observe a Daily Scrum Meeting-up they were well into the swing of things. John, the developer who originally invited

me in was acting as Scrum Master. As with the Scrum Review which I saw earlier, a mix of people from across the company were present.

The discussion circled the table with the Scrum Master asking each person what they did during the previous day. As each person spoke John was navigating the bug tracker to find all tasks assigned to them. The status of each was examined and some rolled over to be completed during future days. These discussions were given focus by the three questions:

- What did you do yesterday?
- What are you doing today?
- Are there any impediments to your work?

There was a real problem getting anyone to admit that anything was an impediment. This might be a feature of an immature Scrum team. Chapter 6 follows an experienced team through a stand-up meeting. Impediments are more freely shared there – and advice is offered during the stand-up. The details of how these interactions are constructed in the mature team is examined in detail in that Chapter.

As each person's work was discussed other people stepped in to pick up tasks which they are better able to handle. Other tasks change status from *pending* to *completed*. The atmosphere of the meeting was open and co-operative and as workloads reduced new tasks were defined, or discovered, and assigned. Justin, the product manager who was so confrontational when I first arrived was especially good at finding, prioritising and sharing work – including taking new things on himself. I noticed, though, that this led to an uneven spread of tasks so that the product manager ended the meeting with lots to

do whilst some of the QA staff had far less work. This was remarked on and everyone just accepted it. I noted at the time "presumably this fluxes over time".

5.9 Discussion

Z* had concrete problems which needed to be addressed. Their communication patterns were a mess: staff were talking to customers without control or oversight and software was continually delivered at the last minute. Because their processes had become chaotic they didn't have the time, space or capability to take on new work. These are all common features of small software houses but at Z* the management reflected and decided to do something to improve their approach to work.

They always tried to be plan-driven. The project management liked to know what the staff were doing, what projects were in-progress and what new work was going to arrive from clients. The use of spreadsheets, backlogs and thorough documentation of projects meant that they had the capability to see a detailed view of all work that was in-progress now and for several weeks ahead. None of this data helped them to run an efficient, low-cost process, instead they constantly made last minute changes and were always fire fighting bugs.

The company took a prudent approach to introducing Scrum, initially implementing it for just one team. They brought in outside expertise in the form of a Scrum Coach to show them how they could make method work on their projects. As they learned to become Agile they followed the Scrum method to the letter. Doing this meant that any problems which they encountered

early in the process were likely to be caused by their understanding or by the way in which they were using the techniques. Staff attended training courses, starting with the senior developers, product managers and project manager who would have leading roles in the implementation. Careful preparation and training meant that the introduction of Scrum need not be haphazard although it still had the potential to bring unexpected problems or changes. The Scrum Coach was in the office for the first week, attending planning meetings and stand-ups. He was able to guide the team through the process, helping them overcome the obstacles which inevitably arise from such a large change.

The biggest changes were not around process, rather they were in the personal relationships between colleagues. Because of the structure of the Scrum the product managers, developers and QA staff were integrated into a single team. The product managers became proxies for the customers, all communication between development and customer had to be routed through them. The product managers also acted as product owners in the Scrums, attending planning meetings, retrospectives and some daily stand-ups. This changed the dynamic of the team so that no-one needed to feel that they were left out of the loop. Both developers and product managers were able to understand the pressures which they each faced.

Part of the promise of agile methods is that improved communication can bring improved process and improved product. At Z* the staff began to communicate more openly and more successfully once they started to use Scrum. Improved communication resulted from increased communication centred around Daily Scrum Meetings at which staff were able to see each others' problems and the efforts which were made to solve them. They began to exhibit the

features of a community of practice as a group of people bound together by shared expertise in a joint enterprise. Cooperation around their work nurtured and sustained the community so that the team's culture became one in which cooperation was at the heart of their working practice.

The initial Scrum trial which is described here was so successful that even before it completed Z* had rolled Scrum out to another project. In the months after this initial effort they gradually switched all of their working practices to Scrum. As their experience and confidence grew they were able to be more flexible about how they use Scrum so that now they can pick and choose the length of a sprint, the number of story points delivered in a sprint and even the regime of meetings which a team has to follow. As they used Scrum they gradually learned about themselves and their customers and came to an approach which means they are better able to respond to customers' needs however fast those change.

This fieldwork revealed that Z* had two significant problems which agile methods might address. First there was internal conflict between departments caused by a lack of understanding of each others' needs and exacerbated by poor specifications. Secondly the developers worked with little or no documentation which meant that, at times, they struggled to understand the code on which they were working.

The ethnographic data which was gathered during this fieldwork was immensely rich. It revealed detailed information about the company, their processes and relationships between the staff. The access to members of staff and to their meetings meant that key steps in their development lifecycle could be followed. The fieldwork at Z* was initiated as a study of their organisational

changes and, consequently samples of talk were not gathered. The subsequent field studies would include both ethnographic data from field notes and detailed samples of talk which would be analytically linked to show both the developers' revealed understandings and the context within which they were produced.

In Chapter 6 a daily meeting is examined in detail, revealing how talking about both software requirements and implementation shares understanding across a team. Chapter 7 follows two developers as they work with existing code to understand and test it and to develop a replacement and shows how talk can reveal more detail than might ever be formally documented.

A*.com 6

6.1 Introduction

Talk is at the heart of Agile Methods. Their power comes from increasing both the quantity and the quality of communication within development teams, and from that communication being talk. When developers discuss the code which they are designing or writing they reveal their understanding, or misunderstanding, about both the problem on which they are working and their solution to it. Detailed conversations are opportunities to share knowledge with other team members or between teams. Talk about work at work can be so useful that Scrum codifies it into a number of ceremonies.

The daily stand-up is the best known Scrum ceremony. In a stand-up meeting the team of developers, product owner and Scrum master meet for fifteen minutes to talk about the work they have done, the work they are going to do and any impediments to their progress.

This Chapter uses a mixed-methods approach, informed by discourse analysis, to examine a daily stand-up at a small software house. The meeting takes place with most of the developers in the same office but the Scrum Master joining in via Skype. Transcriptions of the meeting, taken from audio record-

ings supplemented by contemporaneous field notes are used as the basis of an analysis which reveals how developers talk to their colleagues about their work.

6.2 The company

A*.com is a young company based in Sheffield and London. They are building an innovative Web-based information and document manager using industry-standard technologies, primarily based around the Java programming language and the Spring Web application framework. Development work in the company is done using a broadly agile approach with a small team of technically adept developers reacting rapidly to the needs of their customers.

At first sight A*.com looks like a typical modern startup. The company is innovating, is supported by venture capital and employs a small team of talented young programmers. In common with staff at many other startups the developers are keen to use the latest approaches and ideas and the company supports them. The developers are allowed to choose those technologies and techniques which will work best for the team, management do not impose processes upon them. Since the developers are all young and say that they are interested in finding "best-practices" they ought to be working in the ideal environment.

DeMarco and Lister (1999) describe some of the features of an ideal working environment. Ranging from corporate structure through the composition of the team to the physical setting within which developers work it is clear that there are ways of structuring work which support the development of code and ways of doing so which make it much harder to produce a high-quality

product.

Organisations can create development environments which enable and support programming but this often means that programmers need to manage their own workloads. Glass (2006a) cites studies which show that "... productivity was higher when developers set their own schedule" and "highest when there was no schedule". When I talked to them about their approach to development the developers at A*.com said that they were energised by the ideas behind agile development and chose to use Scrum to structure their work.

6.2.1 The product

The basic premise which underpins A*.com is that many people live hectic lives in which they are constantly balancing the competing demands of work, family and leisure. People are constantly connected to the Web and it is there, using a series of complex Web applications, that they manage their lives. These people buy car insurance on-line, choose and arrange holidays on-line, pay their bills on-line and even communicate with each other using on-line social networks. Shirkey (2008) suggests that for many people their on-line lives are as important as their lives in the physical world.

A*.com is based around a single, simple idea. Life would be slightly, but appreciably, easier if much of the complex data one requires to negotiate the modern World were readily available in a single place. Better yet, mundane tasks such as searching for car insurance quotes could be automated and added to that store of personal or family data. And if the data store had some intelligence it could tell the user when they need to check car insurance quotes because their renewal is due shortly.

A*.com has evolved from that simple vision. Whilst it remains a product that is aimed at families for home use it is not directly marketed. The A*.com strategy is to sell their product to organisations such as insurance companies, utilities or banks and to have *them* offer it to their own customers. This change has had a profound effect on the daily work of the developers. Previously they were building a single product which was a complex enough task, now they were building an infrastructure which has to be tailored to meet the requirements of each client.

Their Sales Director talks to potential customers to understand their needs, he writes a specification which is given to one of the developers who then builds and tests a prototype implementation. This prototype is then demonstrated to the potential customer. A*.com's theory is that showing a working prototype is likely to have a more beneficial impact than showing static mock-ups and will lead to increased customer satisfaction. Deemer et al. (2010) describes how the repeated sprint structure of Scrum keeps a team focused on delivering product to the customer: "[a] key idea in Scrum is inspect and adapt. To see and learn what is going on and then evolve based on feedback, in repeating cycles. The Sprint Review is an inspect and adapt activity for the product". At the end of each sprint the team reviews their progress and how much product they managed to deliver.

Because the company is being driven by customisation and sales, the development process has had to become more responsive. Each new request has to be acted on, there being relatively little negotiation between Sales and Development, typically within a short time frame. The prototypes form part of a negotiation. Once customers see the application's functionality they begin to

think of other things they would like to have it do. Deemer et al. (2010) shows that customers' ideas lead to new requirements which have to be implemented in the next version of the application.

6.2.2 Approaches to development

The continual pressure from new or prospective customer meant that development at A*.com was always iterative. As well as new functionality the developers were constantly revisiting existing code to alter or improve it. When this fieldwork was undertaken A*.com were nearing the completion of a major re-implementation of their back-end system in which they moved from a custom infrastructure to an industry standard one. This change would, they hoped, bring major benefits to them: the custom framework was no longer in development, and its replacement is widely used and known to be both robust and scalable.

The work of this small team had three major drivers. Implementing prototypes for prospective customers, adding or modifying code used by existing customers and improving their infrastructure. That would be a lot of change for an experienced team but most of the developers at A*.com were in their mid-20s. The developers at A*.com wanted to do a good job for their customers and for the company. During preparatory sessions before the fieldwork they talked about using techniques which let them deliver better products for their customers, comments which mirror some of the ideas of the Software Craftsmanship movement. McBreen (2001) writes that "[c]ustomers want great software. Software craftsmen want to produce great software..." and that this "fundamental alignment makes for a better relationship between

customers and software craftsmen".

When changes are made to the code which the customer sees or which impacts upon them changes often also have to be made to the infrastructure of the application. Infrastructure code is at the core of any complex application. The software which users see and with which they interact is often created by a fraction of the code in the application. The infrastructure includes security code which performs authentication, authorisation and message encryption, databases and message queues alongside those sections which implement fundamental business processes. As the users' requirements change or evolve the infrastructure code which supports them has to evolve. This may mean that existing code is modified or that new code has to be added. If a project were run along "fundamentalist" agile principles infrastructure code would only be written when it was actually required. Many project teams, including the one at A*.com, implement a significant proportion of their infrastructure at the start of the project. This has to be done because even the simplest client will require access to functions such as those which provide data storage, networking and so on.

6.2.3 The staff and their working environment

The development team was small and cohesive. The lead developer, software architect and project manager was Dan who lived in Stevenage and usually worked from home. Dan wrote the original version of the software and still *owned* much of the code base. When complex technical decisions were needed the other developers included Dan in their discussions and would often deferred to him. The others claimed that Dan understood everything that was

being done on the development side of the project.

Gary, Ben and Ed did most of the development work. They graduated within a couple of years of each other, had worked in a number of companies and were skilled software developers. Will, Ben's brother, was the team's tester. It was his job to ensure that the software in each release met the customers' specifications for functionality and performance. The fifth team member was Sam. He was on a work placement year whilst at University and had only been with the company for a few weeks at the time of this fieldwork. Sam was helping Will with the testing.

The team was completed by Evan. He was the sales and marketing part of the team and acted as *product owner*. Evan was not present during the field work which is presented here.

The majority of the team worked together in an open-plan office in Sheffield. The office was in a converted industrial unit which has been redesigned internally to support occupancy by multiple SMEs. The A*.com office was a perfect example of the classic space used by modern startups. They had a single large room with a small separate kitchen in one corner. The walls were unrendered red brick. Small late-Georgian sash windows let in limited natural light.

The developers worked at a group of desks by the windows where they faced each other in pairs. The side of the office away from the windows had a number of tables for meetings or discussions. Whiteboards and flipcharts stood around the room covered in notes and doodles.

Most developers would probably consider this working environment near-ideal. The office was quiet and, even when people were talking, it was large enough that noise need not be distracting. Smaller, quieter spaces tend to be

more conducive to software development than do large rooms of cubicles because programming requires so much focused concentration, (DeMarco and Lister; 1999, Glass; 2006a) However with a small team there would be less noise than in a large office. A*.com's room had character which makes it both friendly and conducive to intellectual activity in contrast to the workplaces visited by DeMarco and Lister in their consulting work which tended to be "noisy, interruptive, unprivate and sterile".

6.2.4 Technologies

The A*.com product was a typical enterprise-type Web offering. They have a Web front-end with the business logic implemented in a number of different Java technologies and running on remote servers.

The A*.com technology stack was undergoing some important changes which affected the teams' work. These changes were almost all in the infrastructure of the code which runs on the server. The client code running inside users' Web browsers had not been affected at this point. Because the new infrastructure was significantly different to the previous one a high-level overview of each is included here.

When writing Web applications in Java, developers use libraries of code which simplify the creation of pages and which simplify communication pathways between Web pages, code which implements business logic and the code which manages data storage. In its original form the A*.com product was built on top of the Tapestry library which is developed by the Apache project. Tapestry provides functionality which helps developers build pages and route messages within their Web applications. Using it developers are able to easily

create dynamic Web applications which respond to the actions of users and the data which they submit from HTML forms.

The A*.com application held its data in an Oracle database which was accessed using an object-relational mapping layer, ORM. Early versions of the A*.com product suite used an obscure custom ORM layer called Stash, which was developed by one person for use in his own projects and was supplied to a few of his clients. A*.com was one these clients. They liked Stash because using it was relatively easy but mainly, they said, because they had direct access to the developer.

Having access to the developer of Stash meant that they were able to get features added to it which met their specific requirements. Using third-party code for a fundamental part of the infrastructure of a project can be dangerous. Whilst the technology may work well it was unlikely to have been tested as thoroughly as one might need meaning that it may contain hidden bugs or inefficient code or that it may not scale to the required size of workload. Development on a product such as Stash can end suddenly and support from the developer or user community can disappear nearly as quickly because people lose interest or move on to other things. If a library does become obsolete the code which uses it will have to be rewritten using a different technology. This is called "technical debt" by Fowler (2009) because it is a cost which will have to borne at some future date.

Using off-the-shelf implementations makes a lot more sense. A company such as A*.com makes money from providing services to its customers. Those customers don't want to have to pay to build or maintain the infrastructure which underpins the applications that they are using. Economically, there-

fore, it makes little sense to either build one's own fundamental infrastructure or to use one from a niche provider unless that infrastructure has long-term benefits.

ORM is no longer the niche requirement it once was. It is an established architectural technique used in many applications and solid, robust implementations are available for many languages and platforms. These often provide lots more functionality than is required for a particular project but they have many advantages over custom code. The off-the-shelf product will be thoroughly tested and debugging, there will be lots of support both commercially and on Web sites and the product will often perform very well. It is also far easier to recruit new developers who know the product and can work efficiently with it from day one.

At the time of this fieldwork A*.com were moving their infrastructure from Stash and Tapestry to a framework called Spring which is produced by SpringSource. The SpringSource Website claims that "Spring is the most popular application development framework for enterprise Java. Millions of developers use Spring to create high performing, easily testable, reusable code without any lock-in". Spring gives A*.com similar functionality to that which is provided by Stash and Tapestry alongside a plethora of other capabilities.

Spring is large and complex. Learning to use it effectively and efficiently takes significant time and effort. A complex product such as A*.com's cannot be easily transferred to use a new framework. Significant portions of the code have to be rewritten to use the new framework and, of course, tested once they are rewritten. Whilst the application is being reworked to use the new framework the older version is still in use. Existing customers need to be sup-

ported and potential customers have to be shown demonstrations of a working system. This means that a small development team has to work on what are effectively two separate products. At A*.com, Ed and Dan are working on the transition to Spring whilst the rest of the team work on the existing code.

6.3 Working practices

A*.com appeared to be typical of a small, modern software house. It appears superficially to be like the workplaces described in Sharp and Robinson (2003 2004), Rosenberg (2007). Their working practices would be familiar to other developers. They used a range of software tools, both proprietary and Free Software, to support their development. In conversation with them before this stand-up they said that they try to be customer-focused, to follow best-practices and try to use "the best" available technologies.

For developers the daily stand-up is the heart of Scrum. The stand-up provides a locus for co-ordination within the team without exposing it to external managerial influence, it "is a time for a self-organizing Team to share with each other what is going on, to help them coordinate", (Deemer et al.; 2010). It is the point at which each team member's work of coding and designing is exposed publicly before their peers. The stand-up is a potentially difficult event for a developer because they and their work are placed firmly in the foreground. Many work situations require detailed management of face and relationships but a stand-up exposes these in a way in which few other situations do. When these meetings prove to be useful to the team it is not necessarily because of their structure or of their role within the project. Knorr-Cetina (1995) described how, in scientific laboratories, collaboration comes from "pervasive

co-operation" rather than from structure.

6.3.1 Being agile

The development team at A*.com used Agile practices whenever possible. Specifically, they structured their work as Scrums, using test-driven development and continuous integration through a Hudson server. Lead developer Dan acted as Scrum Master whilst sales director Evan was the Product Owner.

Although each developer had a specific technical role, Gary said before the fieldwork that they "take tasks that need to be done from Dan or Evan and allocate the best person for the job". The division of work depended upon the availability of what Gary called "resources" by which he meant people. To allocate developers to tasks the team needed to know what each of them was currently working on, what they were scheduled to do next and how long those tasks would take. Tasks were listed in an application called Mantis Bug Tracker. Any member could add a task to the list.

Tasks were assigned to team members in a number of ways. Each team member had special areas of responsibility and would automatically take tasks which fitted in that area, for example Gary took most of the client-side JavaScript development. When there were no tasks from the area one of the developers covered he would be expected to choose something which he could handle and which either "looked interesting" or had a high priority. Finally, Dan would assign tasks which need to be completed. These assignments left some tasks which were medium- or low-priority and which didn't sit in anyone's particular area of interest and which, consequently, tended to languish in the bug tracker.

When a developer was allocated a task they made an estimate of how long they would take to complete it. Workload management through estimation is a part of many modern agile methods. When developers accurately estimate the time each task will take their work can be better organised and they can give meaningful schedules to customers. Estimation takes a certain degree of professionalism, needs regular feedback and is a learned skill. As developers spend more time estimating they ought to get better at doing it. Both McBreen (2001) and Sennett (2008) are clear that practising a craft and its allied skills in a considered manner will lead to an improvement in those skills. Estimating is no different to coding in this respect. The more that one does it, the better one will become at it, *provided* that one reflects upon previous estimations. At A*.com Dan worried that their estimates were "off". Using Scrum would, he hoped, formalise the work cycle and, over time, help them become better estimators.

Buglione and Abran (2007) write that estimation in Scrums is done through "experience/analogy" whilst Benediktsson and Dalcher (2003) suggest that setting parameters within which to estimate can improve accuracy. The structure provided by Scrum, especially the stand-up meetings, begins to establish parameters because work is so heavily time-boxed and is considered on a daily basis. Estimates are always contingent and dynamic. Steindl and Krogdahl (2005) write that "[p]lans are only as as good as the estimates, that the plans are based on, and estimates always come second to actuals".

At A*.com the team didn't want accuracy but they did want *better* and more usable estimates. One problem which the team said they had identified when estimating through bug tracking software was a tendency toward the short-

term or quick-fix. Where the development of a solution to for a problem or the completion of a task would take a long time, for example three weeks, tended to be left for later. The same thing happened to those tasks which were assigned a medium priority in Mantis.

The move to Spring from Stash was intended to provide a partial solution. Industry-standard libraries such as Spring tend to be used in standard patterns. Most experienced developers who have used Spring in a commercial project will be able to work on A*.com's new code whereas they would struggle to find developers who could be productive with the Stash library in a reasonably short time frame.

The A*.com team hoped that adopting an agile approach, structuring their work in iterations, using the Scrum methodology and using standard technologies and approaches would help them to address many of their problems.

6.3.2 Skype

This small, distributed team communicated with each other using frequent Skype calls. Even before they became adherents of agile methods they were in constant communication. Voice over IP tools such as Skype can radically change working practices. These tools bring much of the functionality which was envisaged by the CSCW community into any office. Palmer and Fields (1994) listed a number of features and technologies which were required to realise workable CSCW systems including support for distributed synchronous working. using modern applications distributed teams can both communicate and work concurrently on shared desktops.

The A*.com team made extensive use of Skype's instant messenger func-

tionality. They would setup new *chats* each day and use them throughout the day to exchange fragments of code, documentation or thoughts about either code or design. Before introducing Scrum they tended, Gary said, to have one-to-one calls "no more than two or three times a day". Unless they had issues to fix or adding new functionality the team didn't feel that they needed to constantly talk to each other. This changed if they were writing code or tests which impacted on "other people's stuff".

Skype screen sharing was rarely used because they would share information via IM and they used a common code-base in a shared repository using an Apache Hudson server. However it proved useful when testing applications or when demonstrating interface ideas to Evan.

6.4 The stand-up

The stand-up which is analysed here started at 9:15. This meeting took place over Skype. The researcher sat in the A*.com office listening to the meeting and taking notes. The Skype call was recorded for detailed analysis.

The call was initiated by Gary. It was slightly delayed because of a problem dialling the researcher due to the use of different versions of the Skype client. Once that hurdle was surmounted everything worked well.

Scrum Master Dan chaired the meetings remotely. Scrum's daily stand-ups are managed through a set of informal rules which create a rigid structure. Deemer et al. (2010) describe the inner workings of an idealised stand-up meeting. The meeting lasts less than 15 minutes and is attended by the whole team. Deemer et al. (2010) says the key feature of a daily stand-up is that "each member of the Team reports three (and only three) things to the other mem-

bers of the Team: (1) What they were able to get done since the last meeting; (2) what they are planning to finish by the next meeting; and (3) any blocks or impediments that are in their way... There is no discussion during the Daily Scrum, only reporting answers to the three questions".

The stand-up's structure simplifies the Scrum Master's job. In this case the meeting was especially smooth – all of the participants are used to participating in this type of meeting and are very experienced users of conference calls. The meeting worked just like a face-to-face stand-up: each person had a few minutes to talk about what they did yesterday, what they're working on today and what impediments they faced. Dan broke the "rules" of a Scrum stand-up by asking questions and, at times, giving advice.

The stand-up is partially a technical forum, partially an exercise in project management and partially a daily ritual in which the developers bond as a team. Rising and Janoff (2000) write that daily meetings "serve a team-building purpose and bring in even remote contributors, making them feel a part of the group and making their work visible to the rest of the team". The daily stand-up is performing two tasks which go beyond normal project management. It is, they claim, building a team and at the same time it is foregrounding the work of individual team members for comment by line-management and colleagues.

The focus of the following analysis is on the Daily Scrum as a forum for the discussion of technical matters but inevitably the participants interleave talk about both work and personal matters. Talk is rarely just task-focused, it almost always involves the management of identity and of relationships alongside other work. The varied ways in which they manage their interpersonal

relationships at the same time that they have to manage task talk are analysed here.

6.5 Managing the meeting

Daily meetings are one way through which a group of developers using Scrum can become a community of practice. Through the meetings a development team can create a common language to describe what they are doing, a common approach to talking about their work and a united focus. These will be created naturally as they engage in the meeting's main reporting and coordination actions.

The benefits which agile methods offer acquire a different meaning for each team, in each office, on each project because it is what Berger and Luckmann (1966) call a "socially constructed reality". Each team, with its own interpretation and construction of agility, becomes a unique, ever-changing community of practice.

6.5.1 Taking turns

The Daily Scrum Meeting's structure is designed to facilitate rapid progress through the meeting and to ensure that all members of the team get a chance to talk. This is a more complicated task in a conference call than it is in a face-to-face meeting. The Scrum Master must ensure that all of the participants get to speak, that everyone gets to talk as much as they need to and that team members don't get distracted when not speaking. This Section shows how Dan managed this call.

This meeting began with some humorous banter during the set-up phase. When everything was ready Dan brought the humour to an end and began the formal meeting. Table 6.1 shows this happening.

8	Gary	`Ideally he doesn't want us to be all professional pretend like we're a real company he want us to you know laugh`
9	All	*laughter*
10	Gary	`Call each other cocks like we usually do`
11	All	*laughter*
12	Dan	`la la la ok then (0.7) errm did Evan ever show up (.)`
13		`[no]`
14	Ed	`[doe]sn't look like it (.) no`
15	Gary	`[no]`
16	Dan	`good work Kippers`
17		`right then let's errm go round the table (.) let's start with Gaz↑`

Table 6.1: Starting the first turn

At the start of this excerpt, in the *pre-business* phase of the call, everyone was laughing – we'll shortly see why. Dan had to compose himself and get everyone else's attention. He did this by singing a few tuneless "las". The start of the Scrum was signalled when Dan said "OK then". Dan was using this *structuring move* to simultaneously steer them to task talk whilst maintaining a level of informality within the meeting.

Dan knew who was in the call because he could see the list of participants in the Skype application but he checked with everyone that Evan was not there. His approach was not to ask for more information on Evan's whereabouts. Enquiring further about Evan would have placed Dan into a formal managerial position which might be seen as inappropriate in a stand-up in which he was acting as the Scrum Master. Instead he said "errm did Evan ever show up?". Evan, sales manager and Scrum Product Owner, had said he

might be present but hadn't made it to the call. He was not the only missing person: Ben, one of the developers was also absent.

Dan briefly paused after his question before answering himself. Gary and Ed both responded although in slightly different terms. Ed's response was less forceful. He knew, of course, that Evan was absent, but his "doesn't look like it" modified the weight of his "no" so that its impact was greatly reduced whilst distancing himself from information about Evan's absence.

Holmes (1984) demonstrated that people use at least two different strategies to modify the impact of their statements. *Boosters* are used to increase the force of a statement, *downtoners* are used to reduce a statement's impact. These strategies are often implemented using *paralinguistic* devices – pauses, changes of pitch and the use of vocalised "non-words". Ed was using just such a downtoner. Holmes writes that "[t]o reduce the force of an 'unwelcome' speech act is to express positive feeling towards the hearer". The *unwelcome* aspect of this speech act was Evan's absence from the call. Dan was asking, perhaps rhetorically, for confirmation, that confirmation was not really welcome. In addition Dan could see from the list of participants in Skype that Evan was absent, by asking the question he made clear to the other participants, and to the researcher, that Evan's absence had been noted.

Dan's remarks and the replies by both Gary and Ed form a conventional *adjacency pair*. At this point the team are following a conversational turn-taking structure. Later the team's talk will change to be a series of monologues in which they each talk about their work. Before the stand-up formally began they were talking socially. Dan's question about Evan was a turn which, being the first part of the adjacency pair, required a response. The question could not

easily be left unanswered because to do so would be to leave it hanging over the rest of the meeting as something which had to be addressed. In fact it got three responses: Gary, Ed and Dan himself all answered. Dan was answering himself whilst Ed and Gary responded because the question was asked by the more senior member *and* because he was asking it in his role as Scrum Master which gives it some importance in the day's business.

Dan's evaluative move "good work Kippers" reflected his position as an authority figure to the rest of the group. However, the words with which he made this move were very informal. The use of such an informal tone by Dan was typical of this call and of other conversations at A*.com. Informal language such as the use of nicknames was seen throughout the call, and in most of the interactions between team members. The Daily Scrum Meeting is a formal *ceremony* but this team oriented to this formality by maintaining a conversational approach and tone. Fairclough (2001a) describes formality as being generally "a contributory factor in keeping access restricted, for it makes demands on participants above and beyond those of most discourse, and the ability to meet those demands is itself unevenly distributed". If the A*.com team oriented to the meeting as a formal event their discourse would be less free-flowing. Fairclough writes that "discourse in a formal situation is subject to exceptional constraints on topic, on relevance". The format of the Daily Scrum already imposes a restriction on the scope of the team's talk, using formal language would restrict their relationships. In a meeting such as this there is, Fairclough goes on to write, "an exceptional orientation to and marking of position, status, and 'face'; power and social distance are overt and consequently there is a strong tendency towards politeness".

Dan selected Gary to take the first turn. His lead into this selection was, once more, informal. "Let's errm go round the table", on line 40, is conventional and inclusive of his colleagues. Starting with "let's" shows that Dan is asking them to join him rather than telling them to do so.

Moving between speakers

Table 6.2 shows how Dan structured the Scrum to switch the focus from one developer to the next. It includes the end of each turn and the start of the subsequent one.

Each transition is directive but in an informal manner. The etiquette of this point in the Scrum is interesting. This team is quite experienced in using Scrum – they had been following the approach for a year at this point and Ed had worked in other shops which exclusively used Scrum.

When Dan moved the meeting from one member to another he usually placed a clear *boundary marker* rather than making a more complete *closing move*, (Coulthard; 1975). At the end of Gary's discussion Dan said "Okey dokey", he finished his own segment with "errm so that's me", line 91, and Will's segment ended with Dan saying "OK cool", line 115.

Ed's segment of the meeting has been a discussion between himself and Dan. By the end of Ed's segment they had reached a common view and the meeting moved on when Ed signalled his agreement with Dan by saying "yep sure". As Table 6.3 shows the point of agreement is reinforced as Ed mirrors Dan's "with Gaz yeah".

Each turn within the Daily Scrum Meeting is either something akin to a monologue in which the developer talks largely without interruption or a di-

		Handing from Gary to Ed
34	Dan	We can JIT it
35	Gary	We can do something else just too late I mean just in time yeah
36	Dan	Yeah [*laughs*]
37	Gary	[*laughs*]
38	Dan	Okey dokey Ed
39	Ed	OK yeah I was fiddling around springificating all the insurance stuff
40		Errm (1.8) so yeah spent a lot of time annotating classes and turning that thing into a contribution (.) the err primitive mapping stuff
		Handing from Dan to Will
91		(1.3) Errm (2.0) >so that's me (.) let's see what< Will did (1.1)
92	Will	So err yes'day I started of helping Sam with his waters err (.) didn't really take very long doing that 'cause he's getting (0.5) fairly confident at doing them himself [now]
		Handing from Will to Sam
114	Will	[I'll just check] it's still working now
115	Dan	(1.2) >OK cool< (.) Sam
116	Sam	(1.1) Errm (0.4) yeyesterday I was working on one water test pretty much all day which was the (.) errm complete car insurance quote one (0.2) which is (.) slowly gettin' there (0.5) errm (0.8) had to make lots↑ of interesting definitions and things (1.0)

Table 6.2: Moving between turns

84	Ed	Ok yep sure *laughs*
85	Dan	(2.0) And then you're going on to (0.2) stuff with Gaz yeah
86	Ed	With Gaz yeah yeah
87	Dan	(1.6) >Fine< (0.4) <Ok> I'll add what I was doing as a developer

Table 6.3: Moving from Ed to Dan

alogue between the developer and the Scrum Master. In this meeting the former structure was seen when Will reported and the latter when Ed reported, these turns are analysed in more detail in later Sections. There is an expectation that each member will deliver a coherent report, the Scrum Master may interrupt if, for example, more detail is required, but there should not be much discussion – that is reserved until after the meeting-up.

The Daily Scrum has a set of normative "rules" determining the structure and scope of talk within it. These rules are derived both from the generally accepted structure as defined in training materials and literature which is aimed at professionals, and from the culture of an individual team. Stubbe et al. (2003) describe these normative rules as providing "a reference point for participants to treat actions as unremarkable or deviant; participants justify actions as following shared rules or as accountably violating such rules, complain about other's violations, apologize for their own violations, etc". By learning and applying the team's rules, members are able to participate fully in the meeting.

6.5.2 Ending the meeting

The end of the call shows a special version of turn-taking. Because Daily Scrum Meetings don't have space for discussion the Scrum Master has to end the meeting, let the developers return to work and ensure that discussions are going to take place for any impediments which came up during the meeting.

Table 6.4 shows the end of this meeting.

In his role as Scrum Master, Dan asked if "[is] there anything else anyone wants to add?" At this point everyone has had the opportunity to say all

177	Dan	(1.4) Is there anything else anyone wants to add (0.5) other than Ben at magic
178	Sam	(0.3) There probably was several things that I meant to say and I've forgotten (.) oh who wants tea and who wants coffee↑
179	Gary	(0.3) We'll sort that out after
180	Dan	???? of a scrum call
181	Gary	Unless you want to go and give Dan his tea *laughter*
182	Dan	(2.4) Thank you dear Ok
183	Gary	Job done
184	Dan	Job done go go code team

Table 6.4: Ending the call

that they need to and the meeting ended. Sam was self-effacing and took a socially subordinate role when he offered to make drinks. In asking this question Sam demonstrated limited understanding whilst Gary's response highlighted Sam's status within the team. As a new member of staff, Sam had yet to learn the cultural rules at A*. Throughout the call he made a number of other interventions which were inappropriate in this setting.

The explicit end of the call came from an agreement between Dan and Gary that Gary initiated. On line 183 Gary's "job done" was a suggestion that they had completed the business of the meeting. This suggestion was taken up by Dan who used it as a prompt to bring the meeting to an end. Throughout this meeting the atmosphere had been casual and its close was no exception. Dan was willing to be prompted by Gary without needing to react to it as a challenge to his authority.

6.5.3 Authority

The Scrum Master is present as a peer of the rest of the team not as a manager. One of the reasons that the daily Scrum is possible is that it is not a forum for

reporting to management. Rising and Janoff (2000) writes that daily stand-up "meetings address the observation made by Brooks: *How does a project get to be a year late? One day at a time.* When the team comes together for a short, daily meeting, any slip is immediately obvious to everyone. The meetings involve all team members, including those who telecommute".

Dan had to walk a fine line across his three roles. He was a team member, he acted as Scrum Master and, primarily, was in charge of software development. Although managers are meant to be passive observers if they attend a Daily Scrum Meeting, Dan's diverse roles meant that he could not merely observe. He had to be an active player in the Scrum. This tripartite role which Dan had may explain why there is more discussion than reporting in this meeting. The discussions came naturally from talk between colleagues about their work, adjourning all of them until after the call would be unnatural. In particular, Dan and Ed were working on the same pieces of the program. When Ed described the problem he was having, Dan responded in detail although it was "wrong" to do so in a stand-up which follows Sutherland and Schwaber (2007) rules.

Dan balanced his two management functions adeptly most of the time. As Scrum Master he controlled the meeting so that each of the others had time to report their work to the group and to identify any impediments they faced. As senior developer Dan used the meeting to engage with some of the technical problems his colleagues had without demonstrating overt control over their work.

When Dan talked to Gary they interacted as peers. Dan accepted Gary's explanations almost without comment. For example, Gary was experiencing

31	Gary	I mentioned it Evan and he said well if it's gonna work in the short term [we're not]
32	Dan	[yeah] there is that I suppose
33	Gary	we haven't got millions now↑ he says just get it done now and then obviously later on when we do get loads more (1.0) users and files we can we can try it
34	Dan	We can JIT it

Table 6.5: Delaying the search for performance

performance problems with his code. His response to these problems, given in table 6.5 was to out them on one side to be dealt with later when they had more data on the system. Dan not only agrees, he suggests jokingly that they "JIT it", meaning that they solve the problem just-in-time or when they need to. In a more managerial frame Dan might have wanted Gary to work on performance sooner rather than later.

107	Dan	[I] can tell you if he has I don't think he did (0.5)
108	Will	I got the impression he didn't but he thought maybe (.)
109	Dan	Err pages (0.2) sign up (0.2) reset (0.2) there's no activation page in there
110	Will	OK so I won' I won't even bover looking the activation page then
111	Dan	(1.0) No I think↑ just check that it still works because err (0.2) that's the only page left on public that's tapestry driven (0.4)

Table 6.6: Dan searches for a file

When it came to Ed, Dan was much more engaged with the discussion. The talk shown in Table 6.8 showed Dan and Ed talking about the detail of a problem they had each independently discovered. In talking to Ed, Dan was able to make helpful suggestions because he had been working on the same pieces of code himself. During Will's turn, shown in Table 6.6, Dan was helpful, looking for a file on a server for him before advising him about how

to proceed.

146	Sam	(0.2) *Blows out* they're broken
147	Dan	(1.1) They're broken↑
148	Sam	<Yep> (1.4) err utilities is broken on live at the moment or was (0.8) two days ago↑ last time I looked at it
149	Dan	Well it's not going to fix itself (0.5) err are mobile phones in or
150	Sam	Oh I do need to yeah have a look for stuff like that but I need to do (0.4) errm some autocompl>ete< stuff (1.0) for the autocomplete fields (.) so that it can get the value out of it (2.0) which is going to be fun (.) because they don't have a UID
151	Dan	(3.0) Fair enough I'm sure↑ you can work it out (1.4) err
152	Sam	I'll get Will to help me work it out

Table 6.7: Dan encouraging Sam

Table 6.7 contains the end of Sam's turn. On line 149 Dan passed quickly over the broken "utilities", without asking why it was broken or how the problem manifested itself. This contrasts with the earlier detailed discussion he had with Ed but in that case Dan had worked on the same problem. Here he is using a conventional stand-up pattern in which Sam gives an account of his impediments and neither help nor advice is given at the time – doing so is left for separate discussions after the meeting. Talking about *impediments* as Sam is doing is integral to the success of Daily Scrum Meetings. If only successes and working code are discussed then the meeting is left with just a reporting function. By providing a space in which problems can safely be brought to the fore, the meeting lets developers raise issues which can be discussed in detail after the meeting. Here, if one of Sam's colleagues knew why "utilities" was broken they could have talked about it once the Daily Scrum ended.

On line 150 Sam described more of his impediments. Here Dan demon-

strated that he had confidence in Sam, that he could "work it out". This phrase is similar in tenor to his earlier response to Ed when he said "maybe have a look at that". In both cases Dan was showing trust and confidence in the ability of the team members to solve these problems.

6.6 Accounting

The Daily Scrum Meeting provides a very public forum for accounting: each team member must report on their work, and implicitly be seen to be accountable for their progress. The stand-up has a non-evaluative ethos: no-one should receive public criticism or praise although the Scrum Master may praise or criticise privately after the meeting.

The process of accounting for one's work creates a significant threat to one's professional face. The developers' standing and status within the team comes in part from the ability which they are able to demonstrate. Usually conversations are acts of cooperation in which participants work to preserve each others' sense of *face*, (Goffman; 1959). Admitting to failings, for example a lack of skills or knowledge, in front of peers in the stand-up could be embarrassing, (Goffman; 1956). The members in the meeting need to work together to minimise that embarrassment whilst revealing difficulties which may impact on the project. Scrum doesn't have a formal definition of *impediments* which means the word can be used to cover anything each team member wants it to. Some people will raise their personal limitations but most will talk about technical problems or difficulties with the customer rather than using this space to reveal their personal limitations to their colleagues.

Accounting for one's actions in front of peers is hard enough but doing so

in front of one's managers would, for many people, be very difficult. One useful rule of Scrum is that the daily Scrum is not attended by management: it is an event held *by* the development team *for* the development team. The hope is that the team members will not feel that they are being monitored and that these meetings will not fold into staff reviews, appraisals etc. If the meeting is held purely between peers then, the naive assumption is, everyone should be more open and forthcoming. At the start of the meeting there is a *collective intentionality*, (Searle; 1990), oriented towards the meeting's business. As each team member speaks they have individual intentions which may conflict with the wider goals, for example not wishing to discuss a particular problem they are having. The conversation between Scrum Master and team member is initially a negotiation through which they can create a *shared intentionality* which bounds the conversation. The account which a developer provides is necessarily neither neutral nor complete but it will be accepted if the *a priori* negotiated intention was accepted.

The most detailed technical talk in this call was a discussion between Ed and Dan about a problem which they had each independently discovered. The conversation is shown in table 6.8.

Ed introduces the problem on line 49 using terms which distance the problem from him: "when you attempt to go into the service", "doesn't actually exception", "hangs indefinitely". The choice of "you" rather than "I" in establishing the problem creates the maximum distance between Ed and the phenomenon on which he is reporting. But Ed is following the Craftsmanship pattern *Expose your ignorance* which is discussed in Section 3.7.3. The problem must be revealed so that it can be solved and, through the medium of an

49	Ed	Errm so at the moment errm (1.5) bit of a weird one when you attempt to go into the services it doesn't actually ex exception (0.9) but it hangs indefinitely↑ errm so I'm trying to figure out [what's going on]
50	Dan	[I was I] was getting that(0.5)
51	Ed	Right
52	Dan	With some of the stuff I've done errm (0.4) when I was trying to reference a spring bean from tapestry↑
53	Ed	Right
54	Dan	But I hadn't given the spring bean explicitly a name↑
55	Ed	Right (.) ok
56	Dan	And it was just hanging↑
57	Ed	Right ok
58	Dan	and I couldn't work out why until I gave it a name and it worked
59	Ed	OK
60	Dan	So maybe have a look at that↑
61	Ed	Right yeah I'll give that a shot then Err I was thinking it was something to do with Hibern↑ate to be honest 'cause that's the last thing you see
62	Dan	Yeah it's opening a hibernate session m[ine wa]s
63	Ed	[yeah]
64	Dan	And then it just it was obviously somewhere trying to get to a (0.8) bean and failing but I don't know why it hangs it's a bit random↑
65	Ed	OK so yeah figuring that one out really errm
66	Dan	It'll be tha' it'll be fixed in no time
67	Ed	*Laughs*

Table 6.8: A problem shared

impediment Ed is able to do so.

In introducing the problem, Ed provided little information that only people intimately familiar with this piece of code could possibly understand what he was talking about since little context or background are given. Dan was the only other member of the team to have worked on this code and Ed's description was sufficiently indexical that Dan understood both problem and context. Throughout the talk shown in Table 6.8, they talked about Tapestry, Hiber-

nate and spring beans. These are specialised technologies which the rest of the team didn't use and, hence, their colleagues were excluded from the discussion.

Ed's approach to the problem was to minimise his description of its complexity and importance by saying that it was a "bit of a weird one". He made clear that he had the skills to solve this problem: "so I'm trying to figure out what's going on". Dan's intervention was supportive, when he said "I was getting that" he was calling on common experience of the problem and demonstrating that he might know how to fix it. The latter point was manifest in Dan's "maybe have a look at that" on line 60. Here Dan, in his role as the senior developer, was giving a command to Ed but is mitigating it with "maybe".

One of the great benefits of a Daily Scrum is that it acts as a co-ordination point. Here Ed and Dan saw the same phenomenon but had not discussed it until this moment. Without the Daily Scrum they might not have discussed the problem and Ed would have had to spend time working through to find his own solution.

Dan claimed to have made more progress than Ed in solving the problem, discovering that the spring bean has to be given a name. Dan appeared to have found this almost by accident. He said that he "couldn't work out why until I gave it a name and it worked" implying that whilst he hadn't actually found the cause of the problem, he had found that naming the bean removes the error. Both men were happy to have a working solution in this case. Because Dan didn't interrogate his own solution during the meeting he also avoided interrogating why Ed hadn't found a solution of his own. He down-played his own experience and oriented to Ed's professional face needs. On line 66,

Dan moves to reassure Ed about the work. By saying "it'll be fixed in no time" he was expressing confidence that the problem can be solved and, because Ed was going to do the work, confidence in Ed's ability to complete the fix.

The reassurance was important because in their discussion Ed had been reticent to engage with Dan's information. As Dan talked, Ed had given a series of short responses: right, right, Oh OK, right OK, OK, right yeah, which encouraged further explanation. Dan introduced the need to name the bean. Ed's "Oh OK" showed that this was new information for him which bettered his own suggestion that Hibernate as the possible cause of the problem. Dan's clarification that "it was obviously somewhere trying to get a bean" suggests that Ed was looking in the wrong place and ought to have known not to do so.

Throughout this exchange the two men were engaged in discussion which placed them both in potentially face-threatening positions. Ed could not lose face by admitting that he was unable to solve the "obvious" problem, Dan could not lose it by admitting to a solution which he might not fully understand. Dan played down his own status by saying "but I don't know why it hangs it's a bit random" and in so doing he saved Ed's positive face. Ed accepted this, "OK so yeah", and moved the talk back to the usual reporting activities of the Scrum when he said he would be "figuring that one out". Dan was supportive of Ed at the end of this segment in a move which closes the topic.

6.7 Humour

The use of humour is one important factor in the development of a workplace culture and of bonding teams together. Holmes and Marra (2002) write that

humour can make a positive contribution within the workplace by:

- relieving tension,
- counteracting boredom and fatigue,
- energising a discussion,
- and provoking creative solutions and lateral thinking.

The Scrum at A*.com had a number of humorous interludes, often involving either teasing or banter. Usually the response to humour was positive but we present an example in which the humour was rejected by the group. Generally in this group humour fostered a closeness which reinforced their personal and professional bonds.

Holmes and Marra (2002) found similarly well-integrated teams in their study of production-line workers: "[t]his positive picture of a highly integrated and effective team is well supported by the analysis of different aspects of humour in the team's meetings, which suggest that the factory is a lively and verbally engaging place to work, and that a high premium is placed on solidarity and the internal cohesion of the team is reflected both in the amount of humour and the kind of humour evident in team meetings".

6.7.1 Successful humour

Before the meeting began all of the participants had to join the Skype call. Whilst waiting for the final participant (the researcher) Dan and Gary were amusing themselves. They moved their attention to the researcher as shown in Table 6.9.

1	Dan	...because you're rubbish Gary
2	Gary	oh he's in he's in
3	Dan	hooray
4	Gary	yay
5	Dan	is he a silent partner
6	Gary	he is he is he's nothing laughs
7	Dan	so we can say whatever we like about Chris
8	Gary	Ideally he doesn't want us to be all professional pretend like we're a real company he want us to you know laugh
9	All	*laughter*
10	Gary	Call each other cocks like we usually do
11	All	*laughter*

Table 6.9: Before the meeting

This excerpt shows how light-hearted the team could be in the moments before the meeting. At the start of the recording Dan told Gary that he was "rubbish". This would usually require a response of some sort to preserve Gary's face or to acknowledge that he accepts the intended humour. It didn't get a response here, instead Gary was distracted as the researcher joined the call. Gary and Dan mocked the status of the researcher within the call as a "silent partner" and as "nothing" to general laughter. This *levelling* was part of their team culture. By applying it to an outsider they showed acceptance of that person within the group at this time.

The meeting began in a *play frame*, (Coates; 2007), as they transitioned into the core business the play frame was not maintained although there was humour throughout. The stand-up contained one exchange in which a joke was made, shown in Table 6.10. Gary was explaining that he hadn't yet tested the scaling of part of the system but that this would be done when necessary. (Sugimori et al.; 1977) described the manufacturing process at Toyota in which components arrived at the production line as they were needed thus negating

the need for a large on-site inventory. This process became known as "just-in-time" manufacturing. Dan suggested to Gary that they "JIT" the scalability tests, meaning that they do them only when needed or "just-in-time". The just-in-time approach has become so common across so many industries, including software development, that the acronym JIT is often used in place of the full name of the process.

Gary picked up on the idea of JIT and used it to make a joke about the way in which they work when suggested that they do the tests "just too late". So many software projects are delivered late or over budget that programmers joke about doing things either at the last minute or after they were required to. Authors such as Yourdon (2003) and Brooks (1995) have written extensively about the difficulties of delivering software on time. However, Gary was doing something else here, not only joking about software projects which overrun but referring to earlier incidents at A*.com. His use of "else" made telling reference to a history of late delivery which was one of the drivers behind the team's switch to an agile approach.

31	Gary	I mentioned it Evan and he said well if it's gonna work in the short term [we're not]
32	Dan	[yeah] there is that I suppose
33	Gary	we haven't got millions now↑ he says just get it done now and then obviously later on when we do get loads more (1.0) users and files we can we can try it
34	Dan	We can JIT it
35	Gary	We can do something else just too late I mean just in time yeah
36	Dan	Yeah [*laughs*]
37	Gary	[*laughs*]

Table 6.10: Just in time

Dan and Gary were using humour here to subvert the idea of the company

as efficient, effective and business-like. The fact that A*.com did not always deliver on time was not presented here as "a bad thing" since by introducing Scrum the team had done something positive to improve its processes.

Saying outright that the company has, or has had, a problem delivering on time would *usually* be inappropriate. It is something which could be said at a retrospective or at a planning meeting as part of the discussion about workloads, allocations or product delivery which happen in those settings. Saying it as part of the normal day to day conversation at work could be considered impolite as it would reflect badly upon one's colleagues. Holmes (2007) writes that humour can "provide a licence for saying things which would be unacceptable in a serious key". By couching the fact of late delivery in humorous terms Gary was able to make a serious point to his colleagues whilst fostering solidarity between them.

6.7.2 Unsuccessful humour

Humour is not always a successful conversational strategy. Here we look at an attempt at humour which fails.

Sam was the youngest and newest member of the team. At the time this fieldwork was undertaken he had been with the company for a few weeks and may have still been finding his feet both technically and socially and has not yet fully joined the community.

Figure 6.11 shows how Sam introduced his birthday celebration during his Scrum turn. This was an inappropriate thing to say at this time. The convention within a Daily Scrum Meeting is that discussion is to be avoided and the focus of the meeting is on current work. The A*.com Scrum actually con-

119		(0.4) Errm (0.3) I probably should mention now errm I'm not gonna be here on Monday (1.8) 'cause I [boo I t]hink I booked
120	Dan	[hoorah]
121	Ssm	Yeah I booked it off ages ago but then Ben didn't make a note of it
122	??	Oh Ben
123	Ssm	But he says it's fine↑
124	Dan	(1.7) Ok yeah that's useful to know
125	Sam	Yeah 'cause I'm gonna be hung over (0.7) 'cause it's my birthday
126	Dan	[Oh↑ >exciting<]
127	Sam	[on Sunday] and you're all coming
128	Gary	No we're not (.) honestly
129		*General laughter*
130	Sam	Why is the fuel bill fuel bill gonna be too high in your car Gary
131		(2.2)
132	Gary	What↑
133	Sam	Eighty to the gallon it'll do (1.6)
134	Dan	He only lives down the road from you
135	Gary	Yeah why would I drive
136	Dan	Just roll
137	Sam	It's in Derbyshire it's back home [in the motherlands]
138	Dan	[You're giving everyone] an awesome excuse not to go if it's in Derbyshire
139		*laughter*
140	Will?	(2.3) Stay away
141	Dan	Right what you doin' today (0.2) before you err (0.2) jibber jabbered about your birfday
142	Sam	Err I'm finishin' off this (0.4) test

Table 6.11: Going to Derbyshire

tained discussion as well as reporting but these were centred around work, Sam was introducing a personal topic into a work situation. Dan's "Oh exciting" response was a neutral way of moving the discussion away from Sam's birthday.

Gary reply on line 128 was teasing: his "honestly" softened the comment and converted it into a humorous aside. Sam came back with a rather sarcastic comment of his own on line 130 in which he remarks about the quality of Gary's car. In talking about Gary's car, Sam had either missed the humour Gary was essaying or was refusing to play along.Sam was trying to extend this play frame but his attempt was not taken up by the others who instead paused, line 131, before Gary asked Sam to explain what he mean, line 132. Once Sam explained himself, Dan was able to rescue him with a more conventionally amusing comment by suggesting that Gary could "just roll" to Sam's house.

This brief moment shows us a number of things about the team. The obvious point is that Sam did not appear to be integrated into the the team. He tried to make a joke which fell flat with the others who initially made no attempt to come to his aid. Sam was in danger of losing face here. He had made a friendly approach by inviting everyone to celebrate his birthday with him. Sam's response to Gary was pitched badly given that he was the new guy whereas Gary was both popular and a senior member of the development team. Fortunately for Sam, the structure of the stand-up meant that Dan was able to return the conversation to Sam's work when he said "Right what you do in' today before you jibber jabbered about your birfday". This move is both managerial, asking about work, and couched in playful terms. Dan's choice of words, "jibber jabbered" instead of a more formal phrase, reduced tension

and moved the meeting on without a formal assertion of authority (Duncan and Feisal; 1989).

6.7.3 Banter

The use of humour within the team is examined in Section 6.7. Here we consider how the team managed transitions between talk which is not about the task and talk which is oriented directly to the task-at-hand.

Some work situations may have little or no banter with a strong focus on task-oriented talk. The stand-up meeting we are examining here has a lot of informal interactions which need to be managed carefully by the whole group. It is important that the meeting retains its focus because it is intended as a brief co-ordination point during the working day and because they are unlikely to want to have these meetings become too relaxed. On the other hand, stand-ups have an important bonding function within the group and the team's humour serves to strengthen bonds between members.

Plester and Sayers (2007) examined the ways in which IT workers at three companies in New Zealand used humour. Most of the humour which they encountered was described by the authors as "banter", and in the workers terminology as "taking the piss". Banter requires some give-and-take from those involved in it. There have to be two sides to the exchange with each party both giving and receiving some of the humour.

Banter is part of the team's method of relationship management both in this meeting and in their wider approach to their work. Their collective humour bonds them as a team. (Duncan and Feisal; 1989).

When Ed began to discuss his work he used the word "interludes" which

was noticed by Dan and which led to some light-hearted teasing of Ed by the others. Table 6.12 shows this moment.

41		Errm (2.1) in the interludes I started to think about this service for kind of err cloning the database errm (1.6) so writing some notes and started a bit on that
42	Dan	Interludes
43	Ed	Well [like]
44	Dan	[you b]een watching a show or something
45		*General laughter*
46	Ed	No no no the bits where I gave it back to you and you were faffing around with [it]
47	Dan	[*laughs*]
48	Gary	Is that our word of the day↑
49	Ed	Errm so at the moment errm (1.5) bit of a weird one when you attempt to go into the services it doesn't actually ex exception (0.9) but it hangs indefinitely↑ errm so I'm trying to figure out [what's going on]
50	Dan	[I was I] was getting that(0.5)
51	Ed	Right

Table 6.12: A status challenge

When Dan pulled Ed up for "interludes", line 42, he began teasing him. Gary aligned himself with Dan, line 48, by joining in the teasing. Ed does not taken up the humour: at lines 43, 46 and 49 he tries to move them back to task-oriented talk. Dan reverts to task talk on line 50 and the meeting resumes its work focus.

Throughout this Dan was able to maintain his authority as Scrum Master whilst joking with the others. Dan's humour and the subsequent banter within the group act to flatten the organisational hierarchy. Using humour in this way meant that Dan was able to both control the meeting *and* interact humorously with the rest of the team. This was a powerful move on Dan's part.

To successfully make the move Dan had to be confident of both his place in the hierarchy and of the ways in which the others both responded to him and understand his position.

Duncan and Feisal (1989) give four classes of employee who are "over-chosen" or "over-rejected" during humorous interludes at work: arrogant managers, benign bureaucrats, solid citizens and novices. Each of these classes is specially placed when humour is used at work. Arrogant managers tend to sit outside the humorous interactions; novices are the audience for jokes but due to their lower status rarely the butt of those jokes; benign bureaucrats are able to join in with joking but not to make jokes about those who are lower down the organisational hierarchy; and solid citizens are given license to freely joke. Dan was a solid citizen. He was able to initiate the humour and, in so doing, to reinforce his position as a friendly manager. Duncan and Feisal (1989) conclude that " when employees are targeted as butts by managers whom they neither like (the arrogant executive) nor respect (the benign bureaucrat), they take offense. But when a friend and respected peer jokes about them (the solid citizen), the joke is considered to be a compliment". Dan was able to joke with, and about, the rest of the team because he constructs himself as a friendly manager: "[t]rust, respect, and friendship determine a group member's position in the pattern of joking behavior far more than official status does".

The way in which Dan teased Ed is interesting. Asking "you been watching a show or something?" suggests that Ed would not normally use a word such as "interludes". Dan's remarks are a gentle attack on Ed's knowledge. Programmers are engaged in knowledge work: their status comes through their intellectual abilities. Dan was attacking Ed in a vulnerable place by chal-

lenging his intellect.

Ed response includes the mildly derogatory "faffing" which suggests that Ed would have made more progress if it hadn't been for Dan and that he has to challenge Dan's abilities as a programmer. Ed's use of the somewhat derogatory "faffing" showed that he was addressing Dan as his equal. Dan laughed which serves to acknowledge and highlight the humour whilst side-stepping the implicit challenge.

Ed had saved face through his response to Dan's face threat. Dan had been able to maintain his authority. The team showed solidarity by joining in the joking.

6.8 Discussion

The Daily Scrum Meeting meeting provides a structured forum within which members can share information about their work. The meeting's simple structure, the three points which each member has to address, and time-boxed frame mean that discussion is limited. Information can be shared but where there are real problems, for example impediments, detailed discussion has to happen later. However adhering to this structure and to these constraints can be difficult.

The Scrum Master is essential to the success of this event. The meeting which was analysed in this Chapter is constantly in danger of heading off towards one non-work topic or another. That isn't because the team are ill-disciplined or inexperienced, it happens because the meeting has a conversational tone and these topics arrive naturally out of the chatter and banter. Dan uses a lot of skill to keep the other members moving forward. He almost never

asserts his authority explicitly, instead a range of conversational gambits are used to manage the Daily Scrum and his relationships with his colleagues.

This meeting isn't just a place in which information can be exchanged. That can be done using email or IM. The Daily Scrum Meeting which is analysed here shows why this is such an important and effective ceremony for Scrum teams. Within the Daily Scrum, members are able to account for their work, understand the work which their colleagues are doing, find points of common interest, coordinate their work and bond as a team. Each of these helps them to become a community of practice in which shared knowledge, experience and a common approach to their task are embedded and important components of their workplace culture. Through the Daily Scrum, team members develop a shared understanding of what it means to work on this project at A*.com with a set of practices, technologies and approaches which are familiar but which combine into something unique to this workplace at this time.

Work often places people in situations in which they receive unsolicited advice, advice which is face-threatening. Ting-Toomey (1994) shows that threats to face are a normal part of social interactions, but the Daily Scrum Meeting is unusual because it creates a formal space within which they happen. The participants have to preserve their own and each others' status yet the technical content of the discussion may require challenges, both direct and indirect, to other members' status, knowledge or ability. Professionals such as software programmers gain some of their sense of self, some of their self-worth from the arcane and complex jobs which they perform. Any challenge to their competence is potentially a challenge to their sense of self. Even in a close-knit team such as the one at A*.com members have to work hard to ensure that

they work in a cooperative way.

The Scrum Master was accessing the call via Skype. this was not a problem for him or for the rest of the team. He works from home over 150 miles from the A*.com office. The team talk constantly on Skype or on an instant message system. They are experienced with the distancing effect of these technologies and clearly do not worry about them. Once the call had been established everyone talked as if they were in the same room. Again, the structure of the stand-up helps. Unlike normal conversation there are fewer opportunities to talk over other people or to join in conversations. Stand-ups are really dialogues between the Scrum Master and each member in turn. The other members simply listen when it is not their turn. In this call that structure only breaks for moments of banter.

Dan, despite his seniority, was spoken to in much the same way as any other team member, although Section 3.7.3 shows that he was sometimes deferred to for his technical knowledge. Dan demonstrated his skill during this call in handling his roles as programmer, lead developer (and manager) and Scrum Master. At different times he moved between each of these roles, occasionally even performing them at the same time. The others oriented appropriately towards Dan as he changes role throughout the meeting.

One of the notable features of the Daily Scrum Meeting at A* was the volume of humour throughout the meeting. Because this meeting is friendly and conversational, humour is accepted throughout. This demonstrates an implicit aspect of this event: team bonding. The members' use of humour bonds them together as friends. Although the work is taken seriously, as shown by the detailed technical talk found throughout the transcripts, they do not take

themselves too seriously. It may be that, for this team, their humour leavens the face threats which arise in public accounting for their work. Thus, when Sam makes humorous remarks, even though they fall flat, he is both moderating the social pressure on him as he gives his account and trying to orient to community norms and become part of that community. As someone trying to join the community he must serve an apprenticeship in which he learns its norms and finds its boundaries.

Much of the learning of software engineers is not about workplace culture but is about the meaning of the code with which they are working. Chapter 7 follows an experienced developer and a junior colleague as they pair program over a number of days working to understand and re-use some legacy code.

E*: sense-making 7

7.1 Introduction

This Chapter is an examination of the way in which two developers at a small software house work together using the technique of pair programming.

Pair programming is one of the most visible of the many agile practices and methods. The point of pairing is to improve understanding and, hence, to increase the quality of code which is produced. A pair should be better able to understand design documents, existing code and the code which they write because all of those things become part of an on-going discussion. Beck (2000) writes that "if people program solo they are more likely to make mistakes, more likely to overdesign". In a pair this is less likely to happen because, Beck argues, pairing is "a dialog(sic) between two people trying to simultaneously program... and understand together how to program better".

Many researchers have studied both what happens when developers pair and how effectively pairs work. The somewhat nebulous concept of *effectiveness* is typically operationalised in such studies through the measurement of code quality, delivery time or the effort required, (Dybå et al.; 2007).

Results of these studies tend to be mixed, neither conforming nor reject-

ing the hypothesis that pairing makes teams more effective. Madeyski (2007) found that when testing code, pairs were not significantly more effective at fault-finding than were solo developers. In a meta-analysis of those studies which compared pairs and solo programmers, Hannay et al. (2009) found pairing "is faster than solo programming when programming task complexity is low and yields code solutions of higher quality when task complexity is high".

Studies do confirm that pairing benefits developers in their understanding of the task-at-hand. Pairing appears to work because at its core "it is communication, where understanding is developed, agreed and shared", (Sharp and Robinson; 2004). Hannay et al. (2009) conclude that "apparent successes of pair programming are not due to the particularities of pair programming (such as the specific roles of driver and navigator), but rather to the shear amount of verbalization that the pair programming situation necessitates". Sharp and Robinson's idea that understanding is *developed* makes the process appear somehow more structured or controlled than it really is. In fact the pair construct their shared understanding by talking through both the problem and a range of possible solutions. Hannay et al. identifies talk as being the real driver behind any effectiveness benefits which pairing brings but does not analyse the talk to uncover how it drives those benefits.

In these and other studies, collaboration is presented as foremost benefit of pairing and that collaboration arises from communication within the pair. Collaboration is not the same as managing, coaching or tutoring. In a collaboration the two developers are working together, more-or-less as equals, to achieve a single result. Working so closely together a pair of developers will, ideally, share "a substantial amount of visual and mental context", Chong and

Hurlbutt (2007). They look at the same source, develop the same tests, methods etc. throughout the day and, naturally, come to a shared understanding of their work.

Pair programming has interesting dynamics of authority, of sharing and of desire. The metaphor which is often applied to a pair in the agile literature is that of a driver and navigator, (Chong and Hurlbutt; 2007). The division between driving and navigating implies that one of the pair is somehow directing the work of the other. Whilst the roles might be switched throughout the day, the literature does not demonstrate clearly if, or how, the selection of roles by individuals affects code quality or productivity. It is entirely possible that any given pair is more productive when one or the other navigates, for example. One study showed that the dynamic within a pair is far more fluid than might be imagined, and that the dynamic of the pair "differed greatly from the driver and navigator roles described in the academic and practitioner literature", (Chong and Hurlbutt; 2007).

If pair programming is effective because it codifies "communication" as a core practice then studying the communication practices of paired developers ought to reveal how that happens. In many of the existing studies the idea of *communication* is not interrogated, if the term is discussed rather than simply used then a common-sense meaning is applied. What is interesting in this Chapter is not merely that developers working together talk about their work, and that this talk helps them complete their tasks. The real interest is in how they formulate and structure talk so as to share their understanding. The *ethnomethods* through which the developers are able to come to shared understandings are the important matter at hand in uncovering how they work,

(Heritage; 2001, Suchman et al.; 2002).

This Chapter examines what really happens when developers pair, looking at how they communicate about design and about code. This Chapter examines the interactions between two developers who often pair – they "know" how to make pairing work. Their knowledge about the minute-by-minute operation of pairing within their own work is likely to be both tacit and contingent but it will be revealed through analysis of their *ethnomethods*. The two programmers will be followed as they study the architecture of part of a large system, develop new code and tests for both existing and new code.

Data for this Chapter were gathered over a number of days across two weeks. The specific interactions which are transcribed and analysed here took place over three days: consecutive days in one week and one day in the following week.

7.2 The company

E* is a small company which develops mobile telephony applications. Their software is used by businesses such as taxi firms or media companies to send bulk SMS messages to their customers. The applications run on the phone network and on the telecoms company's servers rather than on handsets. E* make a suite of software applications to manage all stages of the messaging process from authoring through the selection of recipients to billing, including comprehensive auditing facilities. It is a matter of pride at E* that, in Managing Director Adam's words "everything should work alongside everything else".

At the time this fieldwork was undertaken E* employed fewer than ten developers with similar numbers of staff in operations and around a dozen in

marketing and sales. The company had customers around the world which meant that they had to make the process of deploying or upgrading software as straightforward as possible both for themselves and for their customers. They didn't have the capacity to spend hours supporting each customer when an upgrade or new version went live. The response to this was to automate processes where possible and to provide good tool support elsewhere. Additionally a number of developments in the operationalisation of software such as test-driven development, continual integration and build and automated deployment are at the core of their processes.

Management and developers at E* described their software development approach as "agile". Rather than defining and restricting their approach to Scrum, XP or DSDM as some agile shops do, E* used a pick-and-mix approach to tools and techniques, selecting those which fitted best with their working practices and ethos. The practices they mainly used were story cards, estimation, pair programming, test-driven development and continual builds. At the time of our visits they were experimenting with Behaviour-Driven Development (BDD), (Chelimsky and Astels; 2010), as a way of integrating aspects of specification and unit testing[1]. The Managing Director, Adam, said that, at times, the company struggled to recruit developers who were happy with its working practices, developers who *can* pair do so whilst those who cannot work in this way leave. Darren, one of the senior developers said a similar thing: a number of people came and went because they couldn't cope with the working practices.

Neither the senior staff, Alex and Darren, nor the other developers found

[1]BDD is an extension of unit-testing in which software is specified through the behaviour it should exhibit.

problems with the automation. Individual developers were keen to talk about the tools and show how they helped simplify build and deployment and, in particular, unit testing. One possible side-effect of the combination of BDD, automation and tool support could have been that fewer staff were needed to achieve the required levels of productivity but this was never mentioned as a problem. The developers presented themselves as a united team which was working towards the same objective of delivering products for customers.

When this fieldwork was undertaken E* developed their applications using a number of Microsoft technologies. Their main programming language was C♯ but the user interfaces to their products tended to be Web applications. When asked about this, a number of team members stated that the agile approaches they had tried worked well for Web development but that they were more difficult to apply on infrastructure and systems-level code. Much of the systems code was written before they began to use comprehensive testing. Adam told me that the backend code "tended to use" integration tests and that they didn't have many unit tests or specifications for it. Unfortunately the majority of code in their applications was just such infrastructure or systems code and the developers had to work hard to apply agile techniques. They said that they went through this struggle because the agile techniques they used allow them to be responsive and dynamic in ways that were much more difficult when using, for example, waterfall.

At all levels the company was very evangelical about the benefits of agility. For example, staff were encouraged to be active in local user groups and to blog about agile methods and the company still runs an active blog. The sales team were encouraged to bring in business which required rapid change to

the codebase, the developers used any approach which worked and even the operations staff used ideas from the burgeoning devops movement, (Humble and Molesky; 2011, Loukides; 2012).

These activities *could* be ones which were imposed by Adam but when asked, several of the developers were active in user groups before coming to E* and others had moved there specifically because of its agile approach. Experienced developers, especially those who were used to more structured or traditional working patterns, find the environment difficult. Consequently E* had tended to recruit only recent graduates it could train in its own approach.

7.3 The development team

All of the staff including management, developers, sales and operations, worked in a single, large, open-plan office. It was a noisy environment – the sales team's phones rang constantly, people had meetings in the corners and the doors slammed each time that someone passed through and, by the water cooler, a radio played indie rock. In many ways it was the opposite of the working environment which DeMarco and Lister (1999) recommend for software development. On the field recordings the noise is a constant mid-level background drone which, unsurprisingly, can become quite disruptive.

The small development team was split into three groups as shown in Figure 7.1. Darren was the Head of Platform which meant that he was responsible for infrastructure, networking and the development of REST APIs; Alex led on billing and applications which were used by non-account customers or for one-off jobs; Neil was in charge of applications which were used by account customers and which were presented as Web pages which had consumed the

services built by Darren's team. The other developers were assigned as teams under the three leads but tended to move around to wherever their skills are needed or to where there was work to be done.

Although the structure seems complex for such a small team it was designed to allow for future expansion – the company were looking for more developers whilst I was there.

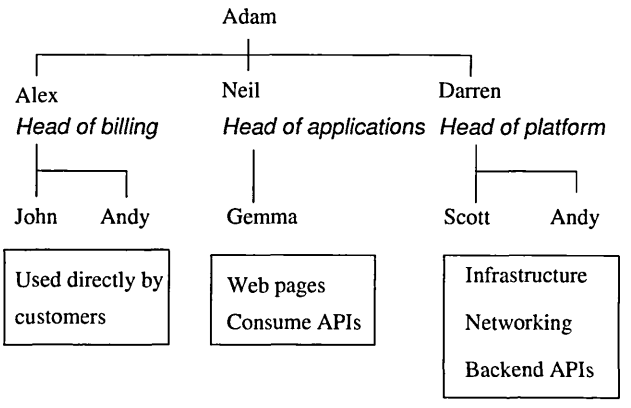


Figure 7.1: The developers' organisational structure at E*

The flexible working practices seemed to be mirrored in their ideas about code. In the words of one E* developer, "code is malleable but design is brittle", approaches such TDD and the use of mocking frameworks let them use the plasticity of code to deliver solutions to customers. Most working software developers have seen situations in which the design is somehow wrong but there is a reluctance to change it. Instead the code changes and becomes out-of-step with the design. At E* both developers and management acknowledged this problem and reacted to it by assuming that the code was always treated as the provider of canonical truths. Later in this Chapter a series of interactions between two developers will be examined. As they try to write tests and implement code they will be seen to use existing code to guide them

instead of referring to design or specification documentation.

7.3.1 Workflow

The team was small and largely self-managing. The Managing Director, Adam, wrote much of the company's original code himself. He continued to take a keen interest in development but did far less hands-on work. His involvement in the detail of the work tended to be at the design stage when customers' requirements were being tightly specified.

The basic approach to development used structural devices from a number of agile methods but would be broadly familiar to anyone who had worked in an agile shop. The following description of their workflow is taken from field notes including conversations with developers and managers.

Each month the team met with Adam to examine their priorities. At these meetings they produced outline plans for the work which they would either complete or begin during the following month. The monthly iterations were short enough that they knew of any new work which was being negotiated so that it could be included in the schedule. The team was rarely so loaded with work that they had problems completing it on time.

Each week the three development leads met with Adam to discuss progress. These meetings were used to refine the monthly schedule, to share possible problems and to identify slack periods. They were able to be both re-active and pro-active in ensuring that work gets done. Each lead was able to consult the other two and Adam throughout the day on technical issues. Although he mostly managed rather than develop, Adam still considered himself to be a software developer and liked to engage with technical discussions. He would

often bring the lead developers into his conversations with the sales team so that sales knew what was being developed and development knew what was being sold.

The team was not only small, they all worked together on a pair of long benches. They talked constantly amongst themselves and helped each other throughout the day. During this fieldwork there didn't seem to be any time at which people were left to struggle on their own. This co-operation was not formalised in the way it might be in a Scrum. Daily meetings were, the developers all said, unnecessary because there was constant talk between them. The constant talk could have been off-putting but if people needed privacy they would wear headphones. Field notes show that when people were interrupted those breaks tended to be brief, for example providing information about some code, and they appeared to quickly re-immerse in their programming. There seemed to be few occasions when people seriously distracted each other. It might be that these workers were used to that environment and didn't let it distract them but equally it is possible that they were distracted but didn't show it. The state of flow which DeMarco (2002) describes enables the most creative work but requires that office are not open-plan and noisy, (Weinberg; 1998, DeMarco and Lister; 1999).

One aspect of the talking was more structured: they liked to use pair-programming when working on especially complicated problems. Very little formal documentation was made available. The code was considered to be self-documenting when read alongside the unit and integration tests. Such tests are not descriptive and the structure of the code only tells you *what* it does not *why* it does it. Much of the meta-knowledge about the code base was

held in the shared understanding of its developers. Collectively they had an understanding of why it was structured as it was, how and why it was meant to work and what its weaknesses might be. On its own the code lacked the *indexicality* which developers might require in order to understand or modify it, (Ronkko; 2007). When the developers *paired* they had to talk about the code and so share their understanding.

XP teams often pair programmers for a day or a week at a time, (Beck; 2000). The E* team modified this practice so that the formation of pairs was dynamic and *ad hoc*. Two might agree that they would work together for half a day or more and would then behave as a normal pair with a driver and a navigator, (Arisholm et al.; 2007). This was not the usual process observed during this fieldwork. Instead people would ask a colleague to pair whilst they worked through a problem, wrote a test or read some code. People would move freely in and out of pairs, assuming that they were not otherwise engaged.

7.3.2 Coding

E* had a lot of code which could be classified in a variety of types. There were stored procedures in the database, Web front ends, XML parsers, code to run the network. There was security code and business logic. And there were tests, many hundreds of unit tests and integration tests. As with many applications this mass of code had grown since the first build of the application by Adam some years before. The code was a mixture of well-engineered solutions, last minute hacks and continual changes. Neither the architecture of the code nor its specific detail were the result of a grand design. Much of it was the product

of gradual accretion as new requirements were specified and implemented or as changes were made to improve functionality or efficiency.

Banker et al. (1998) estimate that over 70% of the "life-cycle" costs of an application occur as maintenance costs, those are the costs of adding or modifying functionality once the software has gone live. In traditional waterfall-style projects there are often attempts to document both the original system and changes made throughout its lifetime. A developer should be able to read the documentation to understand the system before they make their own modifications. In reality developers try to write as little documentation as they can get away with. In defining the first waterfall process for the development of software, Royce (1970) wrote "[a]t this point it is appropriate to raise the issue of - 'how much documentation?' My own view is 'quite a lot;' certainly more than most programmers, analysts, or program designers are willing to do if left to their own devices. The first rule of managing software development is ruthless enforcement of documentation requirements".

The authors of the Agile Manifesto recognised that developers do not like to document their work and proposed "[w]orking software over comprehensive documentation", (Beck et al.; 2001). "Agile methodologies appeared as a reaction to traditional ways of developing software and acknowledge the need for an alternative to documentation driven, heavyweight software development processes", (Ilieva et al.; 2004). All of which begs the question how do developers understand the systems on which they work when the documentation is, at best, sparse or, at worst, inaccurate and out of date?

A range of tools and techniques have been developed in the last ten or fifteen years to help developers handle source code which is either large or

complex. Modern applications are built using more than one programming language and run across a mix of servers, network devices and different types of client. Production systems are assembled from the work of programmers, Web designers, database administrators and security experts. The production version of the code is different to the development version but the two have to somehow be synchronised.

At E* they used a number of tools to help them manage their code. Their development environment was Visual Studio 2008. Using this IDE developers can navigate large projects with relative ease and are able to move between method calls and the definition of those methods with a couple of mouse clicks. Visual Studio supports team working on massive projects yet the same collaborative features help E*'s ten developers co-operate with speed and ease. The team managed the build, test and deployment parts of the lifecycle using a Hudson continuous integration server. The output from Hudson was displayed on a wall mounted flat-screen television so that all of them could see the current status of the development system. The television showed unit test results, build status, and error rates from integration tests alongside charts showing the load on the build server. The on-screen data acted as a near real-time dashboard for the build system. Dashboards can be useful but they have to be used carefully. DeMarco et al. (2008) give a range of good, and bad, practices in their *Dashboard* pattern. The E* build system dashboard presented just enough information to the team that they were able to understand the state of the system without being overwhelmed. Most importantly, the developers were not committing changes to Hudson whilst colleagues were handling bugs or other problems – adding new code at that time would lead to uncer-

tainty and confusion about which changes were causing the problems. The televisions were common around the office. They showed throughput on production systems, status reports for the operations team and sales data. The status of many aspects of the company, not just of software development, was available for everyone.

Being able to move around a code base in Visual Studio is important but the code itself will not tell you about functionality, structure or meaning. Three different testing techniques were being used to help with accuracy and understanding. Integration tests were run throughout the day: each time a change was committed to Hudson almost the whole system was tested. Each module of code was supported by a set of unit tests which defined the functionality of the module as exposed through its interface. The team was trialling Behaviour-Driven Development through a .Net plug-in called StoryQ. In BDD developers write simple, structured stories which describe the behaviours expected from some code. Testing whether the code behaves as expected is an automated process which gives results in Visual Studio alongside the code and the test.

All of the testing practices meant that the team could understand what their code was doing and whether it was doing what they intended. However they had to deal with code which had been developed over a number of years. Some of the original infrastructure code from the first version of their product was still in use. That code was written before unit testing became popular and long before ideas such as BDD became practical propositions with tool support. Older code like this, normally called *legacy code*, always presents a developer with problems, (Ning et al.; 1994, Weide et al.; 1995, Huang et al.;

1996). Firstly, of course, there are the problems of modifying or maintaining this code. Secondly there is the problem of using that code in subsequent functions. As one programs one must make calls into the legacy code but must do so either without confidence about what it will return or optimistically assuming that it works as intended and will produce a valid result.

The interactions which are analysed in this Chapter show two developers working with existing code, struggling to understand it, test it and improve it.

7.4 Gathering the data

Observations at E* were made over a month during Summer 2010. Data was gathered using field notes, photographs and large quantities of audio-recordings of the developers as they worked. The company was extremely accommodating. The Managing Director, Adam, told me that whatever data were needed could be gathered provided the staff gave their permission. Fieldwork was done by sitting with people as they programmed, talking to them about the work, recording conversations and making notes.

Once on site it became clear that the most interesting phenomenon was the pair programming. This led to a concentration on recording situations in which programmers paired together. The data presented here is from just three sessions in which the same two developers work together.

The pair is Darren, an experienced developer and Head of Platform who has been at E* for a number of years and Andy, a recent graduate in his first year at the company. Darren and Andy are an experienced pair who work together often but not exclusively, each will work on his own or pair with another developers depending on the particular need and the availability of a suitable

colleague. In the examples which follow Darren and Andy are trying to understand some legacy code, write some tests and add some new functionality which uses legacy code.

7.5 Working with existing code

Transcriptions in this Section are extracts of a larger transcription which is included as Appendix E.

In the interaction described in this Section, Darren and Andy are re-working some code which was started and abandoned a few months previously. The code synchronises contact lists between the server and a user's phone. Contemporaneous field notes include briefings from the developers about their work and about their specific work tasks. The work they are doing in these transcriptions is framed by the knowledge of the earlier failure and must be analysed through the previous abandonment.

Darren wrote, and abandoned, the code which will be modified in this session. The two men are engaged in a number of more-or-less difficult tasks here. They have to uncover as much meaning as possible from sections of code which can be as brief as a single method call. The indexicality of the source code is a point of negotiation within the pair. They will be seen to try to reach a shared understanding of the functionality they need to use but the code is insufficiently indexical. The context of the production of the original code, implicit relationships between sections of it, and the possibly undocumented role of the code in the wider system must all be understood before the source reveals its meaning.

The differing statuses of the developers provides analytical interest. Dar-

ren is a senior developer, Andy is his junior although he is an experienced and competent developer in his on own right. Within the pair they assume roles as either driver or navigator as described in Section 2.7. These roles may not have equal status since the driver may be more involved than the navigator in the production of new work simply because they have control of the keyboard. The work which Darren and Andy produce depends upon how they interact – if it did not then there would probably be little benefit to pairing.

At the start of the session the two developers are discussing the data which they need to store. Figure 7.1 is a transcription of this talk.

1	Darren	This is kind of the start of a process and over time it might evolve (1.0) or it might not
2	Andy	ha ha
3	Darren	tends to depend how things are used.
4	Darren	Errm (1.8)
5	Darren	yes I think for the implementation side of it the first pass our implementation is about 'cause what we kinda doin' is a provider that's got its own little database here
		(1.5)
6	Darren	tha' that's isolated from that so it can store (.) contacts these'll be probably creatin' it's probably a session-type table
		(1.3)
7	Darren	a user-type table (0.8) ah (1.4) and then the data store
		(1.5)

Table 7.1: Starting the process

Darren immediately takes the lead role by starting the interaction and framing it as "the start of the process" but he doesn't do this in a managerial way of identifying tasks and responsibilities. The field notes show that Darren was one of the original developers of this problem code but not with whom he

worked.

Darren begins the session by outlining his view of what will happen as they work on the code. By saying "kind of" and that the code "might evolve" we see that the solution is, interaction ally at least, open to negotiation. Darren flags that there is a set of possible solutions from which to choose through the use of hedges such as "might", "tends" and "probably". He is doing managing his accountability for the previous failure whilst preserving some status in his dealings with Andy.

It would be difficult for Darren to present himself as all-knowing when they are doing this work to repair a problem with which he was involved. In this light Darren's "or it might not" looks like classic self-deprecation and acts to preserve the face of everyone involved in the earlier work. Andy joins in with some supportive laughter to indicate that he has taken up Darren's self-deprecation and that he understands Darren's turn as humorous. On line 3 Darren continues by making clear that he doesn't expect the code to change unless they need to make changes because of the way that they use it, but he does so in a way which maintains the light-hearted tone which he has already established.

Darren opened up the possibility of the code changing when he said, on line 1 that it might "evolve", however, he is also making clear that only necessary changes are going to happen. In fact the conversation is re-oriented to display Darren's expertise as he says that it "tends to depend how things are used".

Once Darren has made his starting position clear he pauses briefly before he starts to outline what they are going to do on line 5, moving them from dis-

cussion of the existing system by using task talk. He repeats the word "implementation" as he stumbles for a starting point, taking control of the topic and of what is going to happen with the code. As Darren formulates his thoughts through line 6 and 7 he begins to tell Andy what they will do rather than discussing possibilities with him

As he talks, Darren draws a diagram, which is shown in Figure 7.2, saying "what we kinda doin' is a provider that's got its own little database here". Once again he is constructing himself as the one who is expert, the one who does the thinking and has the answers. But he mitigates this by using "we" and "kinda".

This talk is about work-in-progress, about provisionality. This is part of the sense-making structure during this task. The two developers are looking at the code whilst talking about it and its design and deciding upon the immediate future direction which their work will take.

Figure 7.2: The database structure

Using my field notes alongside their conversation I can understand that they are going to develop a service, the provider, which will supply contact details. The diagram is now the focus of Darren's talk, when he says "tha'

that's isolated", he is starting to sketch a database structure. Darren is becoming more specific, pitching to Andy what they will do when he explains how the code works on lines 5, 6 and 7.

Andy takes the conversation in a new direction which is followed in Figure 7.2.

Andy doesn't engage in a long discussion about database structures. The database is simple and he may well assume that they both understand it and don't need to waste time or effort on it. Even so, "what's the sense around using echo?" is an abrupt change of topic to discuss a sketch which they made earlier in the day and which is shown in Figure 7.3. The sketch started as a drawing of the existing, failed, design. It was used to identify the major components and their relationships and in a discussion about the functionality of the system. Our field notes show that Darren uses the drawing as part of pitching when he presents his ideas about the system. He will continue to use this rough diagram over the next few coding sessions as the two men work together to implement unit tests and, later, new functionality.

In asking about the "sense" of the design, Andy's move here challenges that design. Their previous talk has avoided the analysis of problems but here Andy asks for an explanation of a specific choice which was made in the design of the existing code.

Darren pauses when discussing the database design, pauses which are due to his sketching. On lines 9 through 13 the pauses are longer although he is not distracted by any other activity.

Darren responds by saying that the functionality was "kind of" turned off but notice that he hedges. We would expect the functionality to be either on

8	Andy	what's the sense around using echo?
		(1.7)
9	Darren	at the moment that functionality's all kind of been turned off 'cause it it got to a certain point (1.1) and it wasn't really
		(2.8)
10	Darren	to [improve the user experience]
11	Andy	[what was the point of th]at?
12	Darren	the (.) the pla' when we did it to echo (.) then that was (.) the database was here the synchronisation was going and the contacts were pushed up
		(1.6)
13	Darren	into this database. THERE WAS various issues (.) and from a usability point of view it wasn't
		(1.0)
14	Andy	Hmm mmm
		(1)
15	Darren	err >working that< well and also echo (0.9) echo was kind of err
		(1.7)
16	Darren	Yeah (1.0) it had problems (0.3) it was kind of a big
17	Andy	laughs
18	Darren	laughs
19		a big test (.) but this is you know this is kinda tryin' a make it easier. I think as well with that echo side of i' when that (1.1) echo was over here
		(2)
20	Darren	and it was also handling the external contacts so it would kinda get them from salesfo:rc:e so you had to go through >the echo interface to get your contacts from salesforce< whereas we're kind of saying that (0.8) you know (1.5) keeping echo more simple for we're exposing it through the E* API so anyone who consumes this API (.) has then got access to
21	Darren	you know (1.5) keeping echo more simple for we're exposing it through the E* API so anyone who consumes this API (.) has then got access to (1.3) contacts from different ↑sources
		(7.1)
22	Darren	but (0.6) yeah so at the moment it's not really visible in echo.= =I think you ↑CAN?
		(1.3)

Table 7.2: Taking a turn

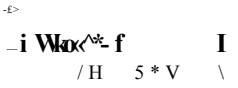


Figure 7.3: The architecture of the system

or off, he doesn't commit to this but places it in a nebulous third state. There is a false start in his explanation, "cause it got to a certain point", before he says that the code wasn't improving the user experience.

Andy interjects during this explanation with "what was the point of that", which seems to ask why the code was removed. This interruption comes at a possible turn transition point following Darren's long delay and overrides the explanation by ignoring it and referring back to his earlier topic. Andy is referring back to the old system once again, pursuing a response and raising questions of both functionality and of the quality of the design and the decision making around it.

On line 16 Darren acknowledges that "echo" was problematic by moving attention to the echo class in a light-hearted way and joking about its state.

By describing it as problematic Darren lets Andy know that they won't be re-using this code. This point is confirmed when he says "it was kind of a big" then pauses. Andy takes up the humour by starting to laugh.

Twice during this interaction Andy asks simple questions which force Darren to explain decisions which were made during the implementation of the earlier code. In both cases Darren has to talk about code which failed in some way but he presents the change as quite positive. On lines 9 and 13 he says that code wasn't working sufficiently well but, notably, he doesn't say that the code didn't work at all. Rather he is implying that changes were needed because the ways in which the code worked could be improved. The desire to improve the code is made clear on line 19 when Darren says "but this is you know this is kinda tryin' a make it easier" and on line 21 with "keeping echo more simple".

When programming simple code structures are often more effective than complex ones because they are clear, more readily "debuggable" and more maintainable. Andy's questions could act as challenges to Darren's authority as the senior developer because they show that even a junior like Andy can spot problems in the code. Throughout this exchange Andy's main contribution is to ask questions or give brief acknowledgements. Darren does by far the majority of the talking. This should not be surprising since they are talking about code which Darren knows and which Andy is seeing for the first time. Andy's questions are requests for clarification and for more detail. In his responses Darren is able to present the design failures which were inherent in the earlier code in a positive light.

7.6 Talking about testing

Transcripts in this Section are selections from a longer conversation which is shown in full in Appendix F.

The day after the interaction described in Section 7.5 Darren and Andy paired again to implement some new functionality based on the work they had started the previous day. One of the key agile practices in use at E* was unit testing. In the session which is presented here the pair begin their coding by writing a test which will be used to demonstrate that the code produces the desired results.

A pair is, theoretically, divided into the roles of driver and navigator. Developers are not usually assigned permanently to a role, they switch between driving and navigating depending on experience, personal desires, fatigue etc. The two roles allow the developers time and space to think about what they are doing without both of them focussing on the physical act of typing code or tests. "The driver is typing at the keyboard and focusing on the details of the production code or tests. The navigator observes the work of the driver, reviews the code, proposes test cases, considers the strategic implications and is looking for tactical and strategic defects or alternatives", (Madeyski; 2007). The practical accomplishment of the driver-navigator duality happens in the talk of the developers.

Testing is seen as an important part of the solution to writing high-quality code. A range of studies demonstrate that good coverage of code with tests can lead to better outcomes, (McBreen; 2001, Kobayashi et al.; 2006, Hanssen et al.; 2009, Hoover and Oshineye; 2010). Of course tests are only useful if they test key parts of the code in realistic ways. Testing peripheral code which

is rarely called using data designed to pass the test might lead to the system generating a success report but it would be a meaningless success.

Tests are created and executed in the programming editor with the results displayed either in red when the code fails the test or green when it passes. In test-driven styles of development a set of tests is written before the code. As code is created, most tests are naturally failed at first but as the code is developed and refined the number of failures reduce until the code passes all of its tests. The test-editor within the IDE is an important actor in testing. It provides a focus, through the red-green messages, for the developers' work. For a pair such as Darren and Andy the editor and its messages are a focus for discussion in the same way as their ad hoc diagrams.

There are no hard-and-fast rules for writing tests. No-one can codify how or where they should be used to give the best possible coverage. Developers tend to learn on-the-job, developing a personal set of heuristics which establish a balance between coverage, the effort required to write the test, and the benefits the test brings. When Madeyski writes of "strategic implications", "tactical and strategic defects" and "alternatives" this is what he means. A pair of developers have the time and space to think about what they will test and why they will test it.

The two men begin by discussing what they will be testing. Table 7.3 shows Darren beginning by nominating some functionality which he wants to test. He starts hesitantly, suggesting one from "a couple of tests". Although test-driven development aims for comprehensive coverage of the code by tests not everything will be tested – pragmatically there have to be limits to the number and complexity of tests which are written.

1	Darren	There's probably there's probably a couple of tests we could ??? here (.) one
2	Darren	err from that premise of that (1.0) ↑a the user might not exist (1.0) so one is (1.0) the credentials are always going to be valid we we kind of (1.0) implement from the perspective that I don't care who who the user is (.) ↑if the user is one I don't recognise I'll create a new one otherwise I'll
		(4.0)
3	Andy	So if ??? do log in with invalid credentials to start off wiv
4	Darren	Well that's the thing I'm saying it's never [going] to=
5	Andy	[Yeah]
6	Darren	=be invalid it will be ↑either one of one that exists or one that doesn't so the scenario the two scenarios as I see it that
		(2.5)
7	Darren	Log in with a (1.0) for the first time as a user and then the test=
		(1.5)
8	Darren	=Is probably going to show something like (.) ah try to get hold to get hold of °this° user from the database the data says it doesn't exist therefore creates a new user in the database and then creates (1.0) a session and returns :that I::d
		(1.5)
9	Darren	And then the other scenario's gonna be where a user °sort of° doesn't exist (1.0) it's gonna call on to the database
10	Darren	and it's actually ??? gets the user the user does all of these
11	Andy	Have you got your diagram

Table 7.3: Setting the scene

It is not clear that Darren knew in advance what he wanted to test when he begins to explain his choice. His approach is to explain what the code is supposed to do which he does by listing constraints on the code: "user might not exist", "credentials are always... valid", "don't care who the user is". These

constraints describe part of the interface to the code without being a comprehensive list and without the detail which would be found in good documentation. Andy's response is to ask about "invalid credentials". His question is a prompt to Darren to provide more detail. Darren has to explain why this code doesn't need to validate the user's credentials and, hence, why they don't need to test them here.

As with the earlier extracts, Darren is doing the vast majority of the talking here and might be said to be "thinking out loud". Andy's main contributions are questions. They are orienting to roles which have nothing to do with either their places on the organisational chart or the ad hoc roles of pairing. Darren presents himself as the expert, he is knowledgeable about the existing code and leads with ideas about where new code will go and about what needs to be tested. Andy asks questions which require explanations and in explaining, Darren has to justify his ideas. Andy is not orienting to the role of a pupil who accepts what he is told, through the questions he demonstrates his own expertise.

This interaction demonstrates how a pair can come to decisions in a different way to two individuals. In the pair they are able to negotiate the "limited indexicality", (Ronkko; 2007), of the code to expose the "strategic implications" of the work they must perform, (Madeyski; 2007). Working individually this exploration would be more difficult because an individual would have had to use the code to reveal its own meaning even where such meaning was unclear.

At the end of this section Andy asks to see the diagram which Darren drew the day before. The diagram was a focus through which they could outline

both the existing system and the proposed solution. Now it becomes a focus for Andy's understanding of the system. Agile projects tend not to create libraries of documentation such as specifications or drawings of the system. Documentation is "always out of date and wrong", (McBreen; 2001), in part because maintaining it is time-consuming and, hence, expensive. That isn't just a problem of software development. Petroski (1996) describes the difficulties engineers at Boeing had in managing the drawings during the design of the 747. Changes to one diagram would have knock-on effects on many others, there were 75,000 engineering drawings for the whole plane which all had to be consistent. When the plane went into production the diagrams were still inconsistent and 1,000lbs of shims had to be used to make components interface correctly.

Engineering companies such as Boeing try to reduce errors in their documentation, and in manufacturing, through the use of process management approaches such as Total Quality Management. TQM helps reduce mistakes through "a high level of cooperation and effort throughout the entire design process", (Petroski; 1996). Efforts such as those of Software Engineering Institute (2010) are building similar process management methodologies for the development of software but they all face a similar problem. Changes to documentation "add no new value to the product", (Petroski; 1996), and the cost of documenting cannot be easily recovered from customers.

The flexibility of pair programming enables cooperation and information sharing without the overhead of formal documentation but developers still need to document. Darren's diagram facilitated both an immediate conversation with Andy and later consideration but it was ad hoc, improvised and

incomplete. As such the diagram was not afforded great value but it did have great utility.

Discussions about ideas and actions allow more consideration than working alone does. Although the pair will implement the tests which Darren would have written had he been working alone he has had to explain them. If he were proposing something which was either inappropriate or incorrect one would reasonably expect that Andy would pick up on this or that Darren himself would realise as he outlined his intentions.

18	Andy	`When do they actually log in then (.) on the phone`
19	Darren	`When the the pho when the phone (0.5) so when you set up the ph[one]=`
20	Andy	`[Yep]`
21	Darren	`=on your phone you'll have some SyncML settings where you'll detail the name of the serv↑er (.) the user name password [which]=`
22	Andy	`[OK]`

Table 7.4:

The field notes show that Andy spent some time looking at the diagram whilst Darren, now driving, began to scaffold the test. Andy then asked for a more detailed explanation of the user authentication process. Figure 7.4 shows how they began this brief discussion which became quite technical as Darren continued.

When Darren has explained the authentication process he asks Andy if he "wants" him to write the test, Figure 7.5. Darren has already created a scaffold framework for the test but now the detail needs to be completed. When Darren asks Andy about his "want" his question is collegiate – he is soliciting Andy's desires and deferring to him – but it is also managerial and controlling. The question "do you want me to do that" on line 28 is used here as an effective way

27	Darren	So at that point it then calls on to the
		(4.0)
28	Darren	do you want me to write the test↑
29	Andy	No I'm just thinking
		(2.0)
30	Andy	So if (2.0) two different mobile phones same user name (.) what happens then
		(3.0)
31	Darren	They're attached to the same u:ser

Table 7.5:

of beginning work on the task whilst recognising Andy's face needs. Darren is ready for them to begin but he is navigating and Andy is driving. Andy has been sitting for four seconds without starting to write the test. If Darren wanted to take over and and simply grabbed the keyboard, he would not be acting aggressively or abnormally. Plonka et al. (2011) found that as many as 81% of role switches were enacted without verbal cues. Here, though, Darren is prepared to wait for Andy to continue.

When Andy replies on line 29 he is both answering Darren and keeping control of the coding task. He tells Darren both that he is going to write the test and that he isn't going to do so yet. When Andy says that he is "just thinking" he is demonstrating his competence as a programmer and as Darren's equal.

Andy's rejection of Darren's offer is glossed over. He is not asked to account for it, perhaps because in the context of a pair programming session with frequent switches such rejection is not going to be an especially accountable event.

The question at line 30 demonstrates that Andy is beginning to understand some of the implications of the processes which Darren has described because he is able to expand the authentication scenario into other areas. His ques-

tion is both reasonable and appropriate and Darren is able to show that it has been thought about and the problem is already solved. This interaction is co-operative. Andy is asking questions which let Darren demonstrate his competence. Darren's answers are structured so as to help Andy to understand, however, Andy always has more questions which often interrogate Darren's assumptions.

For Andy especially, the structure, history, context and quality of the code are always matters which can be interrogated. The code is never indexical of them but, instead, the code provides a locus around which the pair can work towards an understanding. Darren and Andy manage this interaction cooperatively in a way which supports their search for indexicality.

37	Andy	So at the moment we:re here's no such thing as valid credentials then
38	Darren	Yeah that's what I'm saying is that is that the only two two scenarios are [logged]=
39	Andy	[logged]
40	Darren	=in with no cre[dent]ials=
41	Andy	[yeah]
42	Darren	=Or known credentials if it's unknown it goes through a process of creating them (.) if they're known (.) it doesn't
		(1.0)
43	Andy	Which one shall we start with
44		(3.0)
45	Darren	I just say I don't know [it's either its]=
46	Andy	[No no I don't]
47	Darren	=going to be the same ??? name (1.0) do a name cos its natural
48	Andy	hnnn

Table 7.6:

In Figure 7.6 Andy finally gets confirmation about the authentication of users through their credentials. He could have spent time reading through

the code, chasing method calls through the various classes which are involved and looking at the tests written for those methods. He hasn't had to do that because he is in a pair with a colleague who understands what this code is supposed to do.

Andy asks the important question at line 37. Credentials are an important piece of the security infrastructure of any system. Here the user might have credentials or might not, and if they don't then some credentials will be created for them. Some minutes before this interaction Darren had tried to explain how the credentials work. On line 38 he is responding again: "that's what I'm saying" tells Andy that he, Darren, has already said the same things earlier. It is also an acknowledgement, by Darren, of Andy's understanding. When Andy says "yeah" on line 41 he finally signals that he knows what is happening.

At line 43 Andy moves them onto the next stage which is writing the details of the test. He does this without formality, simply moving on to the next phase of the task at hand. In many situations this would be an unusually brusque move but it is more acceptable when in one which is task-oriented. This is another moment which could have become an argument. On line 46 Andy skillfully avoids the possibility of arguing by agreeing with Darren that he also doesn't know where to begin.

By line 52 they are beginning to implement the test, Andy is at the keyboard and begins to explain what he is going to do. It look as if he has taken over the navigating from Darren whilst also typing, but, in reality, the boundary between driver and navigator is a fluid one. The keyboard may not pass back and forth but the talk about understanding does. In Figure 7.7 both men

52	Andy	`I'm gonna make gonna make credent::ials (.) retur::n null what's it return from`
53	Darren	`It returns that's the session ID`
		`(5.0)`
54	Andy	`So this is gonna return null hnnn`
55	Darren	`No I think we are gonna (.) we are gonna`
56	Andy	`Oh no (.) no we're not it's gonna return the session ID`

Table 7.7:

are engaged in understanding what they want the code to do. They need to know what the method returns so that their test can validate the return value. On line 52 Andy has clearly missed that the session ID will be returned but he corrects himself by line 56.

They do not refer to the code in the editor as they talk about it. Each man talks as if his colleague is reading the same thing at the same speed at the same time. The code which they are looking at on screen is the focus for their talk so that if they were reading different things they would have to question each other to find out where the code was that they were discussing.

Throughout the exchange shown in Figure 7.7 Darren and Andy continually interrupt, talk over each other and complete each other's statements. They are familiar with each other's working styles and thought processes. This leads to faster working once they understand what they are going to do. On lines 54, 55 and 56 they Andy quickly understands Darren's intention. By using the same language, in this case the word "gonna", they signal this mutual understanding.

What might be seen as their easy familiarity means that points of potential tension such as Andy's "what's the sense of using echo" in Figure 7.2 do not become major disagreements.

7.7 Implementation or design?

Transcriptions in this Section are excerpts from a longer transcription which is given in full in Appendix G.

Developers who are working with a test-driven approach write unit tests then implement the functionality. The idea of unit tests is that they express the functionality which is required of the code and include the range of results which are possible when the code is provided with different values. The tests are then used to exercise the code. Whilst the tests show *what* a piece of code ought to do, they say nothing about *how* it might achieve those results.

In this Section[2] Darren and Andy are implementing some new code for which they have previously written test cases. Darren is driving whilst Andy navigates.

As is usual with this pair Darren starts the process, shown in Figure 7.8. He begins with a description of some server code which they have already written. On line 3 Darren starts to outline his worries about the way in which the authentication server was implemented. He stops and starts a number of times throughout this line as he talks about different aspects of the management of authentication.

Throughout this passage the pair struggle with uncertainty about how the code they are calling should be used. Until they understand the service which they need to call they cannot understand what their new code should do or how it should be implemented.

They aren't really talking about implementation. They are talking about

[2]This transcription is from a different day to the previous ones, hence the line numbering is restarted.

1	Darren	It's just a simple (.) Do you want me to save the session and err obliterate the database and put it there
2	Andy	OK
		(1.5)
3	Darren	So (.) so that we ohave actually now implemented (.) all of the authentication server so so omy concern that this did <???> was in fact that add session was just literally taking an ID from the user ID but then later we're making calls to the repository to get out some XML based on that session↑ (.) because this decision making process of actually (.) the sync authentication object is not releasing control it's it's deciding I'm going to generate the IDs for the new users=
4	Andy	Yep
5	Darren	=And the new sessions it should also generate a brand new session session=
6	Andy	Yes
7	Darren	=(.) so I'm gonna (.) gonna go back to one of the other tests and change (1.5) so I'm actu actually expecting add session
		(5.0)
8	Andy	I think that the err still still that decision about having these IDs here
		(2.5)
9	Andy	[So now that's doing so much stuff in't it]
10	Darren	[At this stage it's still it's still] yeah bugging me that

Table 7.8:

design, both the design of the service and of the calling code. Implementation requires more detailed talk which refers to algorithms, data structures or the use of specific methods within classes. In software implementations are concrete expressions of ideas, designs are more abstract. A design can be implemented in many different ways. In this sense when Petroski writes about the design of the Boeing 747 he is writing about something which is more like the implementation of software code than it is like the design of that code.

A piece of code has many of the same properties as an engineering drawing has for those who use it. Neither code nor drawing are necessarily fixed or permanent, both are malleable and subject to change as either requirements or understanding improve. When Darren talks about the session being used to get XML from the repository or when Andy talks about code doing "so much stuff" each of them is showing that they now have a different, perhaps more detailed, understanding of their code than they had when it was originally written.

Another layer of abstraction is introduced in Figure 7.9 where the architecture of the software becomes the matter of interest. Software developers use abstraction as a technique for managing complexity when they think about code, (Détienne; 1995, Low et al.; 1996, Hertzum and Pejtersen; 2000). Abstractions are simplified descriptions which foreground parts of the system whilst hiding the details of others. One particularly useful, and commonly used, abstraction is to divide the structure of a system into a series of layers. Each layer wraps lower ones, hiding their detail and providing a hierarchical structure to the code. Layering abstractions in this way is often referred to as the *architecture* of the system.

Figure 7.9 shows the pair using three abstractions at the same time. Darren introduces the system's architecture on line 14 as a way of solving the problem with sessions and user IDs. When he says "that service layer that's responsible for creating the user ID it's responsible for creating the session" he is talking about architecture. The differences between architecture, design and implementation can be subtle and may depend upon the system, the developers and the culture within which they work. Broadly, though, implementation is the

14	Darren	Where this layer (.) actually perhaps this isn't talking to the repository it's talking to a (4.0) service layer and that service layer it's got high level things and that service layer that's responsible (3.0) for creating the user (1.0) ID it's responsible for creating the session
		(2.5)
15	Andy	You're still going to have the same problem though aren't ya (3.0) the same that same you know
		(1.0)
16	Darren	You'll still yeah you'll still get the same problem as when you go down to the [session]
17	Andy	[You're j]ust going to you're just going to copy that method and put it into a (1.0) server side one
		(10.0)
18	Darren	Yeah (.) you c I mean you could have an ID generator
		(6.0)
19	Darren	[Yeah]
20	Andy	[OK]
		(2.0)
21	Andy	Ssss let me have a quick think a second
		(1.5)
22	Darren	I think it is going back again it's that (.) reluctant to do something too clever here if this isn't going in the code
23	Andy	I just say do the simplest thing to start with (1.0) do what you said (5.0) and then it's just going to be hmmm how's it going to kno::w that
		(1.0)
24	Darren	It's literally going to create it is ac I mean it is actually going to create it from scratch

Table 7.9:

detailed of classes and methods, design is how those classes work together and architecture groups classes together into functionally coherent units.

Darren has moved the talk to the level of architecture because talking at this level lets him abstract away the details of the problem. He is able to say that there is a service layer which does what it does and all that Darren

and Andy need to worry about is the result their code gets when it interacts with that service layer. Unfortunately whilst the abstraction removes detail, it doesn't help them to understand or solve their problem. When Andy asks "you're still going to have the same problem though" he makes clear to Darren that he, Darren, has misunderstood and that he is now heading in the wrong direction. Darren reacts positively by agreeing with Andy. The two men sit for ten seconds. That is quite a long time when looked at in the context of their normal working method. Generally this pair take long pauses when one of them is actively coding. Here, though, both are inactive with the opportunity to think about the problem.

Finally, on line 18, Darren offers a tentative "yeah (.) you c I mean you could have an ID generator". The talk shown in Figure 7.9 is cooperative once Andy has steered them onto his track. At lines 16, 18 and from line 22 the two men are aligning their ideas and showing that they are cooperating. They do this to the point where line 24 from Darren follows so naturally from Andy on line 23 that the two statements could have been made by a single person.

25	Andy	So are you thinking
		(10.0)
26	Andy	So that's going to have to [be]
27	Darren	[Well] (.) it doesn't know what the IP is
28	Andy	But how (.) you can't do that can you because that's not going to know
29	Darren	no
30	Andy	That's why we've yeah *laughs* so we either need to (1.0) that shows that's doing too much dun't it
		(6.0)
31	Andy	Can I have a look at that code again

Table 7.10:

As they continue to talk they work closer to understanding what they must

implement. In Figure 7.10 we see they continue to hesitate and to prevaricate. Neither man is willing to make a firm statement of any sort. The nearest they get to a resolution happens on Line 30 when Andy refers back to line 9, Figure 7.8, and says "that shows that's doing too much dun't it". Despite Darren's attempt to change the abstraction up to the architectural level Andy has not changed his view of the code as fundamentally problematic, he tries to persuade Darren by hedging then ending with the question "dun't it". This formulation lets Andy ask a difficult question whilst paying attention to Darren's face needs. Andy's laughter on line 30 and his use of "we" show Darren that they are working through the problem together.

Again there is a six second pause. Darren doesn't disagree with Andy but he is not explicit in his agreement. On line 10 he had said "it's still bugging me that" and on line 16 "still going to have the same problem". In the earlier exchanges discussed in Sections 7.5 and 7.6 Darren was able to provide solutions to any challenges Andy made or to any potential problems that he found. Here that hasn't been the case because they are actively working through code together rather than reviewing and explaining previous efforts. The nearest that Darren has come to responding as if his seniority is challenged is to say that he is "reluctant to do something too clever".

At some point developers need to stop talking about architecture, design or even the structure of code and begin to implement. Figure 7.11 shows this move which is made by Andy on line 33. The move is not straightforward. Before it can be made, the members have to reach a shared understanding, or at least a mutual acceptance, that they are ready to write code.

When Andy makes the move from design to implementation he does so

33	Andy	Or you could just huh well no just a session factory or summat that'll do that that'll get away from that problem won't it (2.0) usually have a session factory >but< all you do is give it a session id
34	Darren	OK
35	Andy	And the user ID and that'll return the session XML back and that way we can make an expecta[tion]
36	Darren	[Session] factory and the user factory yeah
		(5.0)
37	Darren	Do you want delivery receipts and stuff like that (.) just send away where no one goes
		(8.0)
38	Andy	I don't know if we need the user one
39	Darren	Potentially because of this bit as well we could there we could if we're gonna do it then we're kind of consistent
40	Andy	Huh let's start with the session one
41	Darren	Yeah (1.0)
42	Andy	Watch it huh oh yeah
43	Darren	I'll start with the session one
44	Andy	yeah
45	Darren	And see how that goes (.) so I'm going to put a mock in

Table 7.11:

by suggesting, on line 33, that they use a "factory". Factory is a well-known and widely used design pattern in which objects are created without the code which will use them directly calling their constructors. A factory hides the detail of the construction of the object from the code which requires the object by accepting values to be assigned to properties of the instance and returning a reference either to a concrete object or, more typically, to an instance of an abstract base type.

Noticeably Andy doesn't say something straightforward like "let's use a factory". Instead he introduces the merits of the idea, that it will "get away from that problem" and that if they "give it a session ID" and the user ID it will

"return the session XML back". He is talking about design but doing so at the level of the interface to the factory, not saying how it might be implemented. That is enough information to convince Darren that it will work. Andy's approach to getting agreement is to make a suggestion then justify it with more detail.

Once Andy has identified the Factory pattern as the solution to the session ID problem, it also becomes a possible solution to the user ID problem. Darren proposes using two factories on line 36 and formulates this agreement by repeating Andy's talk of session factory. His use of "yeah" at the end of that line confirms his agreement with the use of this pattern. The turns which the men take are collaborative, reflecting that their work here is a collaboration.

On line 38 Andy again questions Darren. He ignores the comment about delivery receipts and returns them to the factory problem. Two long pauses happen between Darren' agreement to the use of factories and Andy's question. Unlike previous occasions, this questioning is hedged by beginning with "I don't know if" rather than a formulation such as "we don't need".

Darren takes this response as a request for some justification of his idea that they use two factories which he then provides. Andy comes back at him on line 40 without a direct agreement or disagreement, instead he grunts at the start of his turn before saying "let's start with the session one" as they begin to code. Andy doesn't say that he has accepted Darren's idea, he just doesn't respond to it. Once again we see that within the pair disagreement is an acceptable part of the negotiation which goes on.

This reply from Andy is a delaying move which implies that they will discuss the user factory once they have implemented the one for sessions or rather

once they "see how that goes", line 45.

Both of them use the "start with" formulation to proceed with the work but in each case it suggests unfinished business, that something must follow the start. The natural suggestion would be that once they have implemented the session factory they will revisit the discussion about a user factory which has not been fully resolved at this point. In fact what actually happens is that they begin to code the session factory and become engrossed in the details of it and that they never return to the user factory. Much later, once they have started implementing the session factory, Darren will suggest that they "let the session factory make the decision about ID" which Andy says "sounds better as well".

7.8 Discussion

These data have nothing to say about the code which is produced from pair programming. Others including Beck (2000), Muller and Padberg (2004), Chong and Hurlbutt (2007) have written about the creation of code by and within pairs. This fieldwork reveals what actually happens within a pair, showing how both understanding and work are shared across time and across tasks.

This pair do not show a strong divide between driver and navigator. Their relative experiences as programmers, and more especially as programmers at E*, impacts on what they know about the code and on how they work. Initially Darren takes the lead, talking Andy through the code or through decisions abut its design. Andy most often acts as inquisitor or prompter, asking for or listening to explanations. This pair differ from others which have been seen to show that "dialogue between pairs was notable for the parity of contribution",

(Chong and Hurlbutt; 2007). Once they move from Darren's explanations to the writing of new code Andy changes his orientation so that by the end they are collaborating with equal status.

Darren spends a large amount of time explaining what existing code is meant to do. The transcripts show that he spends little time explaining how the code works, concentrating instead on its interface and on the data structures which are passed around. Much of the time these discussions are about the system's architecture of the design of the software. Little time is spent discussing the details of the implementation of any of the code. Architectural discussions are aided by the use of Visual Studio which permits rapid, and lightweight, movement between code and tests and within large bodies of source code.

The source is clearly not self-explanatory. Throughout these interactions, which were recorded across a number of weeks, Andy, who is new to this code, demonstrates that he is struggling with the indexicality of the code, a typical programming problem which was identified by Ronkko (2007). The code is an important actor in these discussions, they navigate through directories and files to find relationships within this large codebase but the structure of the code modules and the implementation of individual methods do not reveal their meaning. This appears to be true throughout E*'s large corpus of source code files. In trying to understand what the code *means* the pair constantly move "between the various levels of strategic thinking and implementation detail", (Chong and Hurlbutt; 2007). These movements are only possible because they are paired. Without Darren's existing knowledge Andy would have to spend more time understanding, and without Andy's contin-

ual questionning Darren would be restricted to his memories of the meaning of the code and would not acquire new understanding.

Chong and Hurlbutt (2007) found that "the less knowledgeable programmer instead reported a tendency to become 'passive', disengaging from the task so as not to impede his or her partner's ability to make timely forward progress on the task". That certainly wasn't found in these data. Throughout the coding sessions which were observed at E* both members remained engaged and involved. This was true across pairs regardless of the particular developers in the pair. Feelgood seems to correlate with a willingness to commit to a particular style of working as well to performance once paired as shown by Muller and Padberg (2004).

Pair programming seems to encourage higher quality, and more detailed, discussion of code than is found in, for example, code reviews. When reviewing code, developers who are more familiar with each other or who are reviewing especially complex code have been shown to spend less time discussing "global issues", (Seaman and Basili; 1998). The same study showed that developers who are more familiar with each other report fewer defects during code reviews. When working as a pair talk *has* to move between system wide issues of architecture and structure, characterised by Seaman and Basili as global, and highly localised discussion of in-code structures.

Because the members of the pair are equally responsible for the quality of their code they have an incentive to find and fix defects and to write high quality code where that is possible. When Darren and Andy realise that they need to use factories to create session IDs they are talking about a structure which will lead to better code.

When it works well pair programming can be beneficial. Making it work well requires an openness on the part of the participants. They must be willing to talk freely and be able to talk at different levels, often switching rapidly between those levels. Throughout these excerpts the two men have to manage their talk in such a way as to be able to work together effectively. As the excerpts show they have to be able to interrogate the code, its history and its future. When questioned or challenged they have to be willing to explain or justify their ideas rather than being defensive of them. If they are not open they will struggle to work together.

When working with existing code, Darren and Andy orient to it in a pragmatic way. Andy makes the history of the code, in particular the decisions which underpin its design, accountable. His questions about the code could be taken as questions of practice, of competence. That is not the perspective which is taken by the pair. They legitimise these questions as they discuss the code, making its design into a meaningful part of their discourse.

These interactions demonstrate that, at least for skilled pairs, talk about code is actually more than that. It is talk about history, about design and it is talk which requires skilled management.

Discussion and further work 8

8.1 Introduction

This research is an investigation into the ways in which software developers' communication practices construct and frame their work as programmers. Programming is a challenging activity because the problems it addresses are often vague and because of the complex and plastic nature of software. As discussed in Section 2.5, agile methods address some of the inherent difficulties of software development by emphasising the importance of working code, of teamwork and of cooperation with customers. The authors of the Agile Manifesto placed communication squarely at the heart of their vision of software engineering because they believed that talking about work leads to shared and improved understanding of that work. That may, in turn, lead to the production of better software.

By examining the ways in which programmers talk about programming this research has revealed aspects of developers' sense-making activities and shown how as they share their knowledge they must negotiate meanings and relevance within code and design. The discipline of software engineering can be seen to be as interested in the management of software development as

it is in the activities which constitute development. Agile methods such as Scrum are partly built from the conviction that developers are the best people to manage their own work. This study has shown the communication practices which are implemented in Scrum's cycle of meetings and how developers use these to manage, coordinate and constitute their projects.

This Chapter has three parts: a discussion of the contribution to knowledge of this research; an evaluation of the methodological choices which were made; and the identification of further work which builds on those findings and which can further develop the key research themes.

8.2 Research themes

This research presents studies of interaction and communication within agile teams which are analysed from the perspective of ethnomethodology. This study explored the social organisation of the working practices of software developers and showed that programming is a socially constructed, collaborative activity.

Software development was seen to be a form of heterogeneous engineering, (Law; 1987) in which interactions between developers were as important as the tools they use to develop software. When programming in teams, the developers studied here were shown to share their understanding of code and of libraries, frameworks, legacy code and tools through complex, nuanced conversation. Whether asking questions about code or offering help and advice, the developers negotiated matters of status, of challenges to authority and of threats to their professional status.

The research also found that coordinating work is a fundamental aspect of

the working practice of an agile team. Meetings such as daily stand-ups were shown to facilitate collaborative working when each team member had to report on their progress and status to their colleagues. Once again this reporting process was shown to be laden with potential threats to status which had to be carefully managed by Scrum Masters and developers.

This research began from the idea that the work of programmers is a social activity as well as a technical one. This idea is expressed most obviously in the Agile Manifesto and in those ways of working which it inspires and which are usually called Agile Methods. The Manifesto speaks to more flexible ways of working, which encourage responsive project structures and frequent delivery of working products, and to support them through collaboration and interaction.

Flexible working and frequent delivery have been widely studied, both in the context of agile projects and elsewhere. As discussed in Chapter 3, the ideas of interaction and communication within software projects have received less attention and when they are studied it is usually within an objectivist frame and seeks to find "better" ways of doing them. Neither the authors of the Manifesto, those who implement Agile Methods nor those who try to improve communication and interaction have built a strong theoretical position which explains what they mean by communication and its relationship to the production of software.

The Agile Manifesto is important in providing a context for this research because it places in the foreground the notion that improved communication within teams and between developer and customer will lead to an improved product. However, as discussed in Section 2.6, the meaning of *communication*

and its precise role within agile software projects are matters which neither the Manifesto nor academic software engineers fully address. Reducing communication in software development to talk between the development team and external stakeholders provides a somewhat limited and impoverished view. The communication which happens between developers within teams enables their work of designing, writing and testing code. Although this aspect of the Manifesto may be under-theorised within software engineering, academic work in sociology, psychology and in wider studies of work have, as Chapter 3 discussed, looked at the role of communication in structured situations such as the workplace.

Ethnomethodology provides an epistemological position which explains the place of communication in constructing social order. When the Manifesto's specific principles of "interaction" and "collaboration" are viewed ethnomethodologically, a firm theoretical foundation of accountability and indexicality is available to understand and interpret them.

The questions which this research addresses are, when measured against the normal standards of software engineering, extremely radical. Where most researchers looking at the use of agile methods ask questions about workflows, interaction with customers or the use of tools and technologies, in this research the central questions interrogate the fundamental concept of agility. At a fundamental level these are questions about social interactions and the construction, through those interactions, of specific orientations towards a local culture of agility within teams.

Research into communication within software teams has often been research into ways of engineering "better" communication tools. More funda-

mental ideas about what communication is or the work which is done through communication, as discussed in Section 3.7.2, tend not to be matters of concern in such research. In taking an overtly ethnomethodological position to the study of programming, as this work does, those questions are now matters which are not only interesting for themselves, they become questions which can be used to reveal the working practices of developers.

8.3 Contribution to knowledge

This research contributes to the understanding of the working practices of software developers within agile teams. The most important of the contributions which are made here are those which demonstrate the practical work which is performed by members of agile teams as they implement the ideas of agile methods. The research contributes by:

- showing that developers negotiate the complexities of code not through documentation but by talking to their colleagues,
- revealing how team members coordinate and manage their work collaboratively using daily stand-up meetings,
- showing "the agile move": the set of cultural and organisational changes which accompany the successful introduction of an agile method into the work of a development team,
- addressing aspects of the disciplinary debate within academic software engineering about the status and utility of those empirical approaches and the discipline's dominant objectivist stance.

8.3.1 Negotiating shared understanding

Programmers working in teams have constantly to share information with colleagues and clients. As discussed in Chapter 2, conventional understanding is that the information which is shared within a team about a project comes from requirements documents or the formal documentation of the system. But, as was seen in all three of the companies studied here, statements of requirements are often incomplete or lacking in accuracy and formal documentation is incomplete and out-of-date – if it exists at all.

Programmers spend much of their time working with code which is built on existing code which they need to understand but they often cannot rely on the code to reveal its meaning or on their own interpretations of that meaning as shown in Section 3.5. When (Ronkko; 2007) wrote that developers must strive to find adequate indexicality within code, he was identifying the very plasticity of code as something which can mitigate against understanding. Code is a malleable solution to an, often ill-defined, problem which exists within another domain. Consequently, programmers have to use the knowledge and experience of their colleagues and peers to make sense of the material which they are using.

The knowledge and experience within a team are mediated through the code which is displayed in the editor, through the results of unit tests or, as show in Section 7.5, though ad hoc notes and diagrams. When Darren drew the diagram which is given in Figure 7.2 he was identifying architectural structures which informed subsequent work. Darren and Andy were able to frame their talk by using the diagram to give themselves focus in a way that was seen elsewhere by Suchman and Trigg (1996). The diagram provided a "place" in

which the components within the system could be placed in context and related to each other. In so doing the diagram became indexical of the existing code and of the problems within it.

Typically the code which is written specifically for an application makes only a fragment of the total code in the finished product. Developers are engaged in a process in which they need to understand the users' requirements, create a design for their solution and understand pre-existing code which is in some way the foundations of their own work. Their task is to discern enough of the meaning of designs and code that they can use it correctly in their own applications. In their search for adequate indexicality, Ronkko (2007), programmers are not trying to understand code completely, they are trying to understand code well enough that they can use it.

The meaning of code, its role within a project and its internal structure are matters of interest and import to the developers who are going to use or modify it. Classical approaches to software engineering use documentation as a repository for the understanding which the team collectively has, Sommerville (2004). Such documentation is rarely updated in lock-step with changes to the software, Lethbridge et al. (2003). Agile teams increasingly replace documentation with comprehensive testing, Germain and Robillard (2005), and frequent meetings and code reviews, Seaman and Basili (1998), Deemer et al. (2010).

The importance of team-working as a way of building and sharing understanding was outlined in Section 3.4. (Cahour and Pemberton; 2001) showed that product designers move through repeated propose-evaluate cycles as they collaborate. These cycles mirror both the talk-in-interaction which is analysed

in Chapter 7 and, at a higher level, in the cyclical structure of Scrum.

To some extent the creation of tests from requirements fixes understanding of the external context at a point in time. Teams which use test-driven development write tests based on some expression of the users' requirements, Erdogmus et al. (2005), Hannay et al. (2009). The requirements may be expressed as text documents, as they were at Z*, or as use-cases or user stories but, regardless of form, they are converted into an executable form which can be used by an automated test rig. As the team develop code they use the automated tests to validate their progress: code which passes the tests is assumed to be complete.

But testing and automation simply ensure that the code does what it is asked to do. They do not establish its meaning. To understand what the code does and why it has the structures and features it has, one must either read the documentation or ask the developer and, as discussed, the documentation is likely to be of limited help. Therefore, finding the meaning of, and in, the code is going to be achieved most easily and most accurately by talking to those who developed it or who already understand it in detail.

Software is a plastic artefact whose meaning, structure and context often require negotiation within the team. However, sharing understanding with colleagues is potentially socially awkward. People may not want to ask for help since doing so reveals a weakness either in their skills or knowledge, but sharing one's understanding is difficult if the act of sharing could constitute a face threat for the recipient. If a colleague is asking for information rather then help responding is straightforward since the participants are equals but their relative status is changed when one person asks for help. In that case

the helper is, however briefly, of higher status. The professional status of the person receiving help is necessarily threatened.

Agile projects use a variety of working practices to enable developers to talk about their work to their colleagues. These include pair programming in XP, Scrum's sequence of meetings before, within and after a sprint and the restricted structure of the daily stand-up. Section 2.5 examines some of the core practices of both Scrum and XP.

In Chapter 6 we saw how at A* the daily stand-up meeting moved away from the template of three questions when Scrum Master Dan tackled a problem that Ed was having. Section 6.6 is a detailed examination of their exchange. because the problem was an impediment, Ed had to expose his own ignorance, a pattern which was introduced in Section 3.7.3. Dan's solution involved a change which he had made to some of his own code and which had caused it to work correctly in a similar situation. However, because Ed hadn't asked for help – he was highlighting one of his impediments – Dan's response could be interpreted as carrying a face-threat for Ed. Whilst it is not clear if Ed welcomed Dan's input, he certainly encouraged it through a series of replies which were "right" and "OK". Ed ended the segment by saying that he was going to "figure it out", implying that he would work on a solution even though Dan had just given him a good starting point for one.

At E* the developers work in pairs. A series of interactions within one pair were shown in chapter 7. The two programmers in this study, Darren and Andy, constantly faced the possibility that they would question each others professionalism, skills or knowledge. Darren was more experienced and wrote some code which the two men were using in their work but Andy did

not let his status as the junior partner prevent him asking questions which directly challenged Darren's work. Having the junior partner in a pair challenge the "authority" of the more senior developer is acceptable in Extreme Programming to the point of being explicitly encouraged, Beck (2000).

In Table 7.2 Darren and Andy are shown talking through some code which Andy didn't understand. Andy didn't question how the code worked, he questioned its purpose in the program. Andy is the junior who is questionning the work of his more experienced superior. This is a potential face-threat to Darren to which he responds by working through a long explanation of the code and the reasons why it exists in that place and in its particular form. In so doing he is both explaining and justifying the earlier work.

Most people would find challenges from an inexperienced junior to be either unwelcome or to be implicit challenges to authority and seniority. Weinberg (1999), Williams and Kessler (2000) write of the importance of "egoless" programming in a collaborative situation. A good "pair" will subsume their individual egos to the wider needs of the project, taking reduced ownership of their own code so that they can share in the ownership of the whole project. Researchers such as Bryant et al. (2008), Plonka et al. (2011) have shown that pairing works when the pair are able to think at the same level of abstraction. This was seen in the interactions of Darren and Andy who were each able to talk meaningfully and competently about the code. Extrinsic factors such as their position in the organisational hierarchy or their relative length of employment at E* did not seem to affect how they talked about the work, rather whilst paired they were peers.

This research highlights two situations in which reaching understanding is

made difficult because of the requirement to find indexicality in existing code. In both cases the nature of the code requires that the programmers negotiate meaning through talk and then use their shared understanding as the basis for further work. At A*, Ed and Dan had a problem with a library which Dan claimed to have fixed through experimentation. At the time of the stand-up Ed had not worked through his own solution-finding process and was, consequently, unable to treat equally with Dan. In fact, the problem they had was that a requirement of the library was neither obvious nor documented and both men had mis-used it. Similarly, Darren and Andy were struggling with understanding code on which Darren worked. Darren was able to explain how it worked or why it is present in the program. These two were much closer to their problem code than the team at A* who are using a third-party library which allowed them to talk through a far more detailed understanding.

8.3.2 Coordinating work

The cycle of meetings which form the core of Scrum provide a framework for project level coordination within the team. Work is identified, its relative complexity scored with story points and allocated to developers. Through daily stand-up meetings the team is able to track how well they are keeping to schedule and when there are impediments these are identified quickly so that effort can be put into over-coming them.

At Z* they had a problem with scheduling. Although the managers claimed to know both their resource availability and utilisation most projects ran either late or very close to their deadline. Their development processes were so chaotic that they had no spare capacity for new work but, like so many

medium-sized companies, there was always extra work which needed to be done. The use of automated build systems, bug tracking software and both unit and integration tests are widely identified as best-practice in the industry. Z* used all of these techniques and certainly saw benefits. The testers and developers were happy that both bugs and requirements were available for all projects through the bug-tracker. The product managers felt that they were transmitting their needs to the developers by filing bug reports and that in writing feature requests they were documenting new requirements adequately.

Once Z* moved to Scrum the staff could see that whilst the tools might be useful they weren't helping to solve *their* fundamental issues. Staff at all levels reported that communication problems were foregrounded at stand-up meetings where developers, testers and product managers had to listen to each other talking about their work. In the stand-up which is reported here the staff were reluctant to identify impediments whereas talking to them before their agile move, they were all willing to list endless problems which they felt hindered their work. They reported that this change came through a developing understanding of the problems which each of them faced because those problems were a matter for the daily stand-up meetings and could be talked about as they arose.

The stand-up meeting is a public forum within which team members are displaying their competence and professionalism and their productivity to their colleagues. A key feature of the meeting is that each person reports their progress towards the objectives which they set at the previous meeting. Speaking about the impediments one faces or about a failure to complete a pre-

viously agreed task is going to be face-threatening – especially if colleagues do not find impediments to their own progress. The daily stand-up is run by a group of peers, even the Scrum Master is not someone with significant line-managerial responsibility. There would be a far greater threat to the developers' status if managers were present and if they were to later use the if the result of the meeting when managing staff. At Z* the presence of John, a senior developer, was not a threat. He was not responsible for time-management of staff although he did manage the development process.

When impediments are revealed in a stand-up they can become the collective problem of the group. One incident in a Scrum Retrospective at Z* highlights the change. The team was discussing the work from the previous sprint and identifying a number of problems with the functionality they had been implementing. The product manager was asking questions from the customer's perspective and it became clear that the functionality had not been defined properly in the specification document and was, therefore, open to a number of different interpretations. I showed that rather than blaming each other for the poor specification the team acted collectively to modify the specification so that it could be used in the next Sprint. Everyone I spoke to agreed that before they introduced Scrum, disagreements between the product managers, acting as proxies for the customers, and the developers commonly arose from poor documentation, failed code or missed deadlines. Once they understood something about each others' problems they worked more collaboratively.

At A* the team was as engaged with problems of functionality and code design in their stand-up as with coordinating their work. The stand-up was a

discussion of progress towards objectives which, for a number of team members, meant that it was a discussion of the impediments they were facing. Each person knew where rapid progress was being made and where there were delays. This type of information becomes especially useful at the end of a Sprint when reflecting upon its achievements towards its objectives. The stand-ups provided useful, although undocumented, information about the difficulties which individual developers had during the sprint so that workloads can be adjusted accordingly for the next iteration.

Here the Lead Developer acted as both Scrum Master and line manager. This meant that he was in a potentially compromised position in which he had to balance his managerial role with his role as one of the developers. The structure of the Scrum stand-up helped him because it allowed him to control the meeting so that each of the others had sufficient time to report their work to the group and to identify any impediments they faced. Team members did not have to so struggle for a chance to speak. Instead of negotiating turns amongst themselves, the turns were allocated by the Scrum Master.

Chapter 5 showed how the Scrum Master used his status as senior developer to engage with some of the technical problems his colleagues had. In his discussion with Ed about a problem which they had both seen, Dan spoke as a developer. They talked about the problem as a problem of programming rather than as a problem with Ed's workload or with external impediments. Whilst this wouldn't happen in a Scrum which adhered strictly to the guidelines, it was a modification with which the A* team were happy. In a strict Scrum the need for such technical discussions is identified in the stand-up but the discussions are conducted after the meeting.

At A* technical knowledge could be shared in the stand-up and, even when there was a disagreement, Scrum Master Dan did not take overt control over his colleagues. Whenever he told them what to do he couched his instructions with politeness which minimised face-threats. Dan reacted calmly when presented with problems. Sam had an issue with a broken test. Dan glossed over the detail of the problem and instead told Sam to "work it out". This phrase was similar to the one Dan used to Ed earlier in the meeting when he said that Ed should "maybe have a look at that". In these formulations Dan was directing the other staff but not telling them what to do, instead he was relying on them to work out the details for themselves. Having the trust of senior staff in this way means that people do not have to be afraid to, as one apprenticeship pattern discussed in Section 3.7.3, has it, "expose your ignorance". When face-threats are minimised there is less reputational damage in discussing difficulties which one has with a piece of work.

Away from the ritual meetings of Scrum, programmers are constantly having to coordinate amongst themselves. At both Z* and E* automated build systems were in use. Completed code was checked in to a version control system before the entire development application was built so that a series of integration test could be run. Before check-in the programmers would ask their colleagues if they were using the system and if they were when it would be free. Developers at all three companies would regularly talk to each other about their work either to solve problems or to uncover the work which was happening elsewhere so that efforts could be coordinated.

8.3.3 The Agile Move

Two of the three companies which were studied here were long-term users of agile methods and the research undertaken with them was typical in that it examined how they worked within the broad framework of agility. The study of Z* was different because it was a study of a team as they began to use Scrum. The agile move is an interesting one for a development team since it brings major changes in working practice with it. In the case of Z* pressure to become agile came from developers who wanted to reduce some of the tension and conflict in the company. Not only were conflicts reduced, such major cultural changes came with the agile move that the use of Scrum spread throughout the company to all project teams.

Before Z* made their agile move the Project Manager had been concerned that he would lose control of the work they were doing. He was used to using spreadsheets, Gantt charts and Microsoft Project to develop a holistic view of the activities of all project-based staff. All of this documentation was subject to constant change as customers constantly adjusted their requirements – in effect it was always out of date. The Project Manager's "control" of resources was something of an illusion, he actually spent his time tracking and auditing resources rather than deciding how they could best be used.

Following the introduction of Scrum the product managers, developers and QA staff were integrated into a single team. This had two effects. First the team was largely able to manage its own resources and to make resourcing decisions based upon the current situation. Secondly the relationship between product managers and developers changed significantly as each group came to share the difficulties of the other during their daily meetings.

At Z* the benefits of agile methods were real although not readily measurable. Improvements in communication were reported by many of the staff but those weren't measured in increases in the amount of code produced or reductions in the number of bugs. Intangible benefits were reported: staff understood each other, there was an increase in cooperation and the product managers, sales staff and developers worked more closely as a team.

The team from Z* which was followed here experienced some of the classic benefits of agility. Whilst the developers were keen to experiment with agility, they were cautious in their introduction of Scrum. Staff attended training courses ahead of the agile move and they brought in an experienced Scrum Coach to mentor them. Using a Coach meant that they were able to avoid potential misunderstandings in their implementation. The Coach guided them through the process, in particular showing techniques for running the various Scrum meetings which had been shown to be effective elsewhere.

The Scrum team at A* hadn't used coaching or training when they began. They reported that both senior developer Dan and Ed had used Scrum in previous jobs. Their understanding of the method was shared with their colleagues. The team at E* used an *ad hoc* collection of agile practices picked up from previous jobs or at user groups and conferences.

The Agile Move which Z* made was based on the prior experience of agile methods of their coach. The use of agile methods at A* and E* was, in both cases, simplified because they employed staff who had already worked in that way. Running self-managed, iterative projects is complicated. The project management is not necessarily more complex than in a traditional project, in fact that might be easier, but the processes which are used in agile are diffi-

cult ones. This research examines two of them: stand-up meetings and pair programming. Despite agile's implicit commitment to egoless programming, each approach puts developers into situations which are, potentially, socially and professionally threatening and which require careful interactional management. As this research has demonstrated, part of the skill of Scrum Masters or "pairs" is to manage the threats whilst working cooperatively. The cultural shift which agile brings can, as Z* found, change an organisation. Given major opposition from some senior staff each of those pitfalls could have lead to the cancellation of the Scrum pilot and a return to the tyranny of the Gantt chart.

The detailed sections of Conversation Analysis throughout this research show that conversations in a knowledge-work setting are a delicate balancing act. In daily stand-ups the developers constantly face threats to their status as their approach to their work is exposed to the scrutiny of peers. Two things ameliorate the face threats in these meetings. First the Scrum Master plays an essential role in facilitating the stand-up, ensuring that the meeting runs smoothly by sticking to its simple structure. The art of managing a stand-up seems to be in keeping the meeting on track whilst not becoming authoritarian or overly managerial. Both John, at Z*, and Dan, at A*, were able to move from person to person and topic to topic smoothly and without dissent. At no time in any of the stand-ups which were seen at either company was there a disagreement about the management of the meeting. None of the participants asked for extra time or expressed the need to say more or to return to earlier points.

The stand-up at A* shows the second way in which threat is avoided in a stand-up. The meeting was seen to have a social function in helping develop-

ers bond as a team. At A* the team were mostly at ease with each other, they were able to use humour throughout the meeting. Whilst taking their work and their professional status seriously, frequent banter between team members served to lighten the mood of the meeting. In this respect the team at A* was different to that at Z*. Stand-up meetings at Z* had far less off-topic banter and little or no discussions beyond the work process. The Z* stand-ups adhered very closely to the pattern of those from an idealised Scrum. These stand-ups have many of the features of a Community of Practice. Although the meetings are structured the talk-in-interaction does not have to be and so stand-ups provide a space in which people can "share their experiences and knowledge in free-flowing and creative ways", Wenger and Snyder (2000).

Pair programming is a fundamentally different way of working for programmers compared to their normal solitary work. The purpose of pairing is to share knowledge and to design collaboratively. Using detailed conversation analysis revealed this is what happens case but that talk can be focused through the study sections of code or the creation of notes and diagrams. Chapter 7 showed how the diagram given in Figure 7.3 allowed Darren and Andy to negotiate their understanding of the architecture of the system. The diagrams, code and ad hoc notes which developers such as these use all facilitate the negotiation of meaning. When there is such a focus for talk, it becomes the embodiment of practical indexicality which can be applied to the problem at hand to enable collaborative sense-making.

8.3.4 Qualitative software engineering research

Section 3.8 discussed the place of qualitative research studies in software engineering. The discipline has traditionally placed most value on quantitative work based on experimentation or on engineering novel solutions. Both researchers and practitioners tend to an objectivist approach in which the point of research is to find better ways to build better software. Whether they are writing a compiler or evaluating the use of an agile method the vast majority of software engineering research is quantitative, researchers measure and tally the impact of their ideas on developers. Too often, though, sample sizes are small or the subjects are students who, whilst readily available and cheap, are not representative of the wider community of programmers.

An alternative community of qualitative researchers is growing. These researchers are interested in human aspects of software engineering, in understanding how developers actually behave. Many remain objectivist in outlook. They want to help improve development practices but to do so from an understanding of what really happens when people write code.

This research has shown that qualitative techniques are appropriate approaches to gathering data about professional developers as they manage their work and understand architecture, design and code. Studying developers in their workplace as they perform their daily tasks revealed their situated knowledge and understanding. When staff at Z* talked about problems of project management, for example, they were talking about *their* specific and unique problems on *their* projects. When Z* began to use Scrum the interest lay not in how Scrum might, in the abstract case, change communication patters. Rather, the interest was in the changes in communication which happened for

that group of people on that project at that time.

Software engineering research often searches for general problems or solutions which can be applied widely. Hence the question which people ask about agile methods is more likely to be "how are they applied" than "what problem might they address for this team". But the benefits, or otherwise, of the adoption of any technique are contingent upon the context within which it is used. The same is true of the negotiation of understanding.

At E* Darren and Andy spent a long time trying to understand existing code. They didn't discuss the details of class structures or the signature of individual methods, instead they talked through the *meaning* of the code for their work in their project. Indexicality was seen to be located in the specific detail of their work. The type of situated understanding towards which they worked is different to more general ideas about the meaning of code.

The research presented here shows that qualitative empiricism lets the researcher reveal the located understanding and practices of practitioners. It shows the culture of specific teams working together at one moment in time. Whilst this research is, obviously, not the first to draw out the importance of locally situated workplace culture it does reinforce the utility of fieldwork in understanding how programmers actually work together.

8.4 Further work

This research has demonstrated that the ways in which developers construct and understand their work can be revealed and analysed through ethnography, audio recordings and detailed conversation analysis. A number of other aspects of programming could be profitably explored either by extending this

work or by building on the current findings. In this Section some of the possibilities for further work will be introduced.

Two of the Agile Manifesto's four values place importance on communication by favouring *Individuals and Interactions* and *Customer Collaboration*. The other two values are *Working Software* and *Responding to Change*. The Manifesto's values are important in this work in its themes of examining *understanding* and *coordination*. Agility is embodied in the relationship between customers and developers through which requirements are identified. As customers and developers understand more about each other the set of achievable requirements is refined and may even, by the end of a project, change completely. Researchers have undertaken numerous projects in gathering requirements from, and with, users. Little work has been done to understand how those requirements are interpreted by the people who are implementing them in code. There is scope to use the techniques from this research to study what requirements mean to programmers and how those meanings are made manifest in tests and code which they write.

Each of the cases which were used in this study was of a small, motivated team working for a small company. Larger organisations which have many teams working together on projects have complex coordination needs. When projects are distributed across teams the programmers within those teams have them same need to share information as they would have in a small team but with the added complexities which distance brings. Whilst there have been research studies which examined how distributed teams coordinate their work, few studies have looked at how developers share their understanding of the code which they write or use with colleagues who are in

different locations. In the study of A* in this project one team member was working remotely but he checked in with the rest of the team throughout the day and often attended face-to-face meetings and social events with them and was fully integrated into the team. It would be interesting to see if similar levels of integration are possible in more widely dispersed teams.

The analysis from E* included an examination of a pair of programmers using an ad hoc diagram as a tool for understanding the architecture and use of a piece of code. Diagrams are one of the most commonly used documentation techniques in software development. Programmers have to be able to think about the detail of one part of a problem, or solution, whilst ignoring the detail in the rest of the system. Software is simply so complex that it cannot be thought about without hiding details which are unnecessary at a given moment. Diagrams appear to provide a tool through which complexity can be described whilst the detail of that complexity remain hidden. There have been some studies into the use of formalised notations such as UML but few into the ways in which ad hoc diagrams help programmers manage complexity. Studies which build on the work undertaken at E* would generate new understanding of how programmers manage complexity.

Whilst diagrams are important, programmers mostly locate their understanding of problems and solutions in the code which they write. This research has shown the central importance of code in the working lives of programmers. A common-sense understanding of programming would, of course, identify code as the single most important artefact in the whole process of writing software. The discussions in the daily stand-up meetings at Z* and A* were meant to be discussions about the process of work but often became

discussions about the structure, meaning or complexity of the code. The conversations at E* were all about matters of code. Given the importance of code one might expect that software engineering would place it at the heart of disciplinary interest. Whilst there has been much work on complexity and on language design, much less work has been done to look at how programmers *understand* their code. Expanding this research to specifically study how representations either as code or in diagrams enable shared understanding would be important and useful work.

Finally, many developers have moved to test-driven or behaviour-driven approaches to development. The conversion of user requirements into specifications for tests or behaviours and the use of those to define the interface of code is another interesting area for study. Many research question arise from the use of tests including what it means to a developer to program to a test, how the tests are tested, how tests feed into software quality audits and other approval processes and even what processes are used to test the tools which run automated testing suites on which developers come to rely.

8.5 Conclusions

Software development is sometimes seen as a purely technical activity that appeals to introverts. The central argument of the Agile Manifesto was its observation that if software developers communicate with each other and with their clients they will create better products in a more timely fashion. This research asked three questions:

- how do software teams coordinate their work through their talk-in-interaction

about that work?

- how do software development teams which use agile methods create and sustain an agile culture?
- how do software engineers talk about code so as to make sense of it?

The case studies were of three organisations which had different experience with agile methods and which used different methods in their work. Between them these studies not only helped to answer the research questions, they also demonstrated that software development is a form of heterogeneous engineering. It was shown that when writing software successful developers use a variety of forms of social interaction alongside their technical knowledge and practical competencies to achieve their goals.

Coordinating the work of teams of developers has always been challenging. The discipline of software engineering grew from the idea of a software crisis in the 1960s which was, itself, predicated on developers inability to deliver software on time and under budget. Software engineers responded to the crisis by codifying project methodologies that centralized the control of work in the person of the project manager. Agile teams take the control of their work upon themselves. The studies at A* and Z* showed this happening. At A* an experienced Scrum team used their daily meetings to as a forum at which each member informed the rest of the team about the work which they had done and were planning to do. Although the A* team took a relaxed and informal approach to their interactions, the stand-up was the point in each working day at which they shared information about the progress of the project.

The Agile Manifesto is not only a call for changes in working practices, but also for a change of culture within software development. Perhaps the Mani-

festo resonates with so many developers because it places them at the centre of the development of software. The study at Z* demonstrated the beneficial power which such a cultural change can have. Informants at Z* described it as an organisation in which conflict between different teams, especially between those who were customer-facing and those who engineered the product, were endemic. The introduction of Scrum at Z* had relatively limited impact on the product life-cycle since they already ran to tight deadlines and iterative release cycles. The major impact of Scrum was to improve communication between all of those who were working on a project. In the initial Scrum project the culture changed from one which was grounded in distrust and conflict to one in which people cooperated, in part because they understood the difficulties which their colleagues were having.

Agile methods such as Scrum can change organizational culture in terms of the way in which work is managed but they have less impact upon the practical work of software engineers as they design, write and test code. The programmers at E* used pair-programming as a way in which they could work together to produce their software. Pair programming creates a situation in which developers must talk about the code they are writing. More than that, however, they must share their understanding of both that code and the existing code on which it is built. The production of code at E* was seen to arise from the context of its production through the history of earlier decisions which had created previous structures and functions. However, the historical context was hidden, being neither embedded in the code nor available in design documentation. Context was revealed only when the programmers talked about what the code did, why it did it and why it had the legacy structure it had.

In some ways software engineers are bricoleurs for whom successful work is not simply a matter of technical ability but also of an appreciation of the context of the production of that software. Producing working code which meets the requirements of end-users is possible because software developers, both individually and in teams, can communicate effectively. Agile methods provide structures, spaces and working practices which foster cultures that promote and value just such effective communication.

Bibliography

Noura Abbas, Andrew Gravell, and Gary Wills. Historical roots of agile methods: where did agile thinking come from? In *Agile Processes and eXtreme programming in Software Engineering*, pages 94–103, 2008. URL `http://eprints.soton.ac.uk/266606/`.

Theodore Abel. The operation called verstehen. *American Journal of Sociology*, pages 211–218, 1948.

A Abran, J.W. Moore, P Bourque, and R Dupuis, editors. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2004 edition, 2004. URL `http://www.computer.org/portal/web/swebok/html/contents`.

ACM. Computing Degrees and Careers, 2012. URL `http://computingcareers.acm.org/?page_id=12`.

Zaidoun Al-Zoabi. Introducing discipline to xp: Applying prince2 on xp projects. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–7. IEEE, 2008.

Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1978. ISBN 978-0195019193.

Christopher Alexander. *The timeless way of building*. Oxford University Press, 1979.

Bob Anderson. Work, Ethnography and System Design. In A Kent and J G Williams, editors, *The Encyclopedia of Microcomputers*, pages 159–183. Marcel Dekker, 1997.

Erik Arisholm, Hans Gallis, Tore Dybå, and Dag I. K. Sjø berg. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

Paul Atkinson. Ethnomethodology: A critical review. *Annual Review of Sociology*, 14:441–465, 1988.

Rajiv D Banker, Gordon D Davis, and Sandra A Slaughter. Software development practices, software complexity and software maintenance performance: a field study. *Management Science*, 44(4):433–450, 1998.

Francesca Bargiela-Chiappini. Face and politeness: new (insights) for old (concepts). *Journal of Pragmatics*, 35(10):1453–1469, 2003.

Stephen R Barley. Technology, power, and the social organization of work: Towards a pragmatic theory of skilling and deskilling. *Research in the Sociology of Organizations*, 6:33–80, 1988.

Chris Bates, Kathy Doherty, and Karen Grainger. "What's the Sense of Using Echo?" Social Interaction in a Pair Programming Session. In *RAISE 2011*, Preston, UK, 2011.

Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, first edition, 2000.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Hunt Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development, 2001. URL `http://www.agilemanifesto.org/`.

Andreas Becks, Tim Reichling, and Volker Wulf. *Social capital and information technology*, chapter Expertise finding: approaches to foster social capital, pages 333–354. MIT Press Cambridge, MA, 2004.

Oddur Benediktsson and Darren Dalcher. Effort estimation in incremental software development. *IEEE Software Proceedings*, 150(6):351–358, 2003.

Gabrielle Benefield. Rolling out Agile in a Large Enterprise. In *41st Hawaii International Conference on System Sciences*, Hawaii, USA, January 2008. IEEE.

Peter Berger and Thomas Luckmann. *The social construction of reality*. Penguin books, 1966.

Alan F. Blackwell, Kirsten N. Whitley, Judith Good, and Marian Petre. Cognitive factors in programming with diagrams. *Artificial Intelligence Review*, 15 (1-2):95–114, 2001.

B. Boehm and R. Turner. *Balancing agility and discipline: A guide for the perplexed*. Addison-Wesley, 2004.

Barry Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

Barry Boehm and Richard Turner. Management challenges to implementing agile processes in traditional development organisations. *IEEE Software*, pages 30–39, September/October 2005.

R. J. Boland and R. V. Tenkasi. Perspective making and perspective taking in communities of knowing. *Organization science*, 6(4):350–372, 1995.

Eric Brechner. *I.M. Wrights Hard Code.* Microsoft Press, 2007.

Fred P. Brooks. *The Mythical Man-Month anniversary ed.* Addison-Wesley Longman, 1995. ISBN 0201835959.

Antony Bryant. 'It's Engineering Jim...but not as we know it' Software Engineering - solution to the software crisis, or part of the problem ? In *ICSE 2000*, pages 77–86. ACM, 2000. ISBN 1581132069.

Sallyann Bryant, Pablo Romero, and Benedict du Boulay. Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, 66(7):519–529, 2008.

Erik Brynjolfsson. The productivity paradox of information technology. *Communications of the ACM*, 36(12):66–77, 1993.

Luigi Buglione and Alain Abran. Improving Estimations in Agile Projects: issues and avenues. In *4th Software Measurement European Forum*, pages 1–31, Rome, May 2007.

Monika Büscher. Social life under the microscope? *Sociological Research Online*, 10(1), 2005.

Graham Button. The ethnographic tradition and design. *Design Studies*, 21: 319–332, 2000.

Beatrice Cahour and Lyn Pemberton. Keeping the Peace: A Model of Conversational Positioning in Collaborative Design Dialogues. *AI & Society*, 15: 344–358, 2001.

Jeff Carver, Carolyn Seaman, and Ross Jeffery. Using Qualitative Methods in Software Engineering. *Empirical Software Engineering*, 2004.

David Chelimsky and David Astels. *The RSpec book: behaviour-driven development with RSpec, Cucumber, and Friends.* Pragmatic Bookshelf, 2010.

Sebastien Cherry and Robbillard P. N. Communication problems in global software development: spotlight on a new field of investigation. In *Third International Workshop on Global Software Development*, Edinburgh, UK, May 2004.

J. Chong. Social behaviors on XP and non-XP teams: a comparative study. In *Agile Development Conference (ADC'05)*, pages 39–48. IEEE Comput. Soc, 2005.

Jan Chong and Tom Hurlbutt. The Social Dynamics of Pair Programming. In *29th International Conference on Software Engineering*. IEEE Computer Society, 2007.

CMMI Product Team. CMMI for Systems Engineering/Software Engineering, Version 1.1, Continuous Representation (CMMI-SE/SW, V1.1, Continuous) (CMU/SEI-2002-TR-001). 2001. URL `http://www.sei.cmu.edu/library/abstracts/reports/02tr001.cfm`.

Jennifer Coates. Talk in a play frame: More on laughter and intimacy. *Journal of Pragmatics*, 39(1):29 – 49, 2007. Focus-on Issue: Topics in Applied Pragmatics.

Alistair Cockburn and Jim Highsmith. Agile software development: the people factor. *IEEE Computer*, pages 131–133, November 2001.

Mike Cohn and Doris Ford. Introducing an Agile Process to an organisation. *Computer*, pages 74–78, June 2003.

Melvin E Conway. How do committees invent? *Datamation*, April 1968.

Michael Coram and Shawn Bohner. The Impact of Agile Methods on Software Project Management. In *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*. IEEE Computer Society, 2005.

Malcolm Coulthard. Discourse Analysis in English- A Short Review of the Literature. *Language Teaching*, 8(02):73–89, December 1975.

Mike Crang and Ian Cook. *Doing Ethnographies*. Sage Publications, 2007.

Nigel Cross. Creativity in Design: Analyzing and Modeling the Creative Leap. *Leonardo*, 30(4):311–317, 1997.

Nigel Cross and Anita Clayburn Cross. Observation of teamwork and social processes in design. *Design Studies*, 16:143–170, 1995.

K. Crowston and E.E. Kammerer. Coordination and collective mind in software requirements. *IBM Systems Journal*, 3(2):227–246, 1998.

Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of ICSE 2008*, page 241. ACM Press, 2008.

Pete Deemer, Gabrielle Benefield, Craig Larman, and Bas Vodde. The Scrum Primer. Technical report, The Scrum Foundation, 2010. URL `http://assets.scrumfoundation.com/downloads/1/scrumprimer121.pdf?1294640838`.

Tom DeMarco. *Slack*. Broadway books, 2002.

Tom DeMarco and Timothy Lister. *Peopleware: productive projects and teams*. Dorset House, 2nd edition, 1999.

Tom DeMarco, Peter Hruschka, Timothy Lister, Steve McMenamin, James Robertson, and Suzanne Robertson. *Adrenaline Junkies and Template Zombies*. Dorset House, first edition, 2008. ISBN 978-0-932633-67-5.

M. Denscombe. Communities of Practice: A Research Paradigm for the Mixed Methods Approach. *Journal of Mixed Methods Research*, 2(3):270–283, July 2008.

M. D'Eredita and C. Barreto. How Does Tacit Knowledge Proliferate? An Episode-Based Perspective. *Organization Studies*, 27(12):1821–1841, December 2006.

Françoise Détienne. Design Strategies and Knowledge in Object-Oriented Programming: Effects of Experience. *Human-computer interaction*, 10(2-3): 129–170, 1995.

Jorge L. Diaz-Herrera. The "engineering" of software, a different kind of engineering. *ACM SIGSOFT Software Engineering Notes*, 34(5):1, October 2009.

Yvonne Dittrich and Kari Rönkkö. Talking design. *Technical Paper*, 2002.

Peggy Doershuck. Incorporating Team Software Development And Quality Assurance In Software Engineering Education. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, Savannah, Georgia, USA, October 2004. IEEE.

P. Dourish and G. Button. On "technomethodology": foundational relationships between ethnomethodology and system design. *Human-computer interaction*, 13(4), 1998.

Jack Duncan and J. Philip Feisal. No laughing matter: Patterns of humor in the workplace. *Organizational Dynamics*, 17(4):18–30, March 1989.

Emile Durkheim and Lewis A. Coser. *The division of labor in society*. Free Press, 1997.

Carol S. Dweck. Motivational processes affecting learning. *American Psychologist*, 41(10):1040–1048, 1986.

T. Dybå and T. Dingsoyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, August 2008.

Tore Dybå, Erik Arisholm, Dag I K Sjøberg, Jo E Hannay, and Forrest Shull. Are two heads better than one? On the Effectiveness of Pair Programming. *IEEE Software*, 24(November/December):12–15, 2007.

Penelope Eckert. Communities of Practice. In *Encyclopedia of language and linguistics.*, pages 1–4. Elsevier B.V., 2006.

Kathleen M. Eisenhardt. Building theories from case study research. *Academy of management review*, 14(4):532–550, 1989.

Amr Elssamadisy. InfoQ: Is the Agile Community Being Unreasonable?, March 2010. URL `http://www.infoq.com/news/2010/03/agile-unreasonable`.

Paul M Eonardi. *The Mythos of Engineering Culture: a study of communicative practices and interation*. Master of arts, University of Colorado, 2003.

Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. On the Effectiveness of the Test-First Approach to Programming. *Proceedings of the IEEE Transactions on Software Engineering*, 31(1), January 2005.

Sean Esbjorn-Hargens. Integral research: a multi-method approach to investigating phenomena. *Constructivism in the Human Sciences*, 11(1):79–107, 2006.

Alberto Espinosa, Sandra Slaughter, James Herbsleb, Robert Kraut, Javier Lerch, and Audris Mockus. Shared Mental Models, Familiarity, and Coordination: a multi-method study of distributed software teams. In *Proceedings of 23rd International Conference on Information Systems*, pages 425–433, 2002.

Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

Norman Fairclough. *Language and Power*. Pearson Education, second edition, 2001a.

Norman Fairclough. Critical discourse analysis as a method in social scientific research. In Ruth Wodak and Michael Meyer, editors, *Methods of Critical Discourse Analysis*, pages 121–138. Sage Publications, 2001b.

Samer Faraj and Lee Sproull. Coordinating expertise in software development teams. *Management science*, 46(12):1554–1568, 2000.

Robert R Faulkner and Howard S Brecker. *Do you know...? The Jazz Repertoire in Action*. The University of Chicago Press, 2009.

Karl Flinders. The world's biggest civilian IT project finally looks to have failed but is the NHS IT failure a surprise?, 2011. URL `http://www.computerweekly.com/blogs/inside-outsourcing/2011/09/the-worlds-biggest-civilian-it-project-finally-looks-to-have-failed-but-it-is-no-surprise.html`.

Samuel C Florman. *The Existential Pleasures of Engineering*. St. Martin's Press, first edition, 1976.

Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

Martin Fowler. Technical Debt, 2009. URL `http://martinfowler.com/bliki/TechnicalDebt.html`.

W. L. Gardner and M. J. Martinko. Impression Management in Organizations. *Journal of Management*, 14(2):321–338, June 1988.

Harold Garfinkel. Ethnomethodology's program. *Social Psychology Quarterly*, 59(1):5–21, 1996.

Robert P Gephart. The Textual Approach: risk and blame in sensemaking. *Academy of Management Journal*, 36(6):1465–1514, 1993.

Eric Germain and Pierre N. Robillard. Engineering-based processes and agile methods for software development: a comparative study. *The journal of systems and software*, 75:17–27, 2005.

John Gill and Phil Johnson. *Research methods for managers*. Sage, 3rd edition, 2002.

Barney Glaser and Anselm Strauss. *The Discovery Of Grounded Theory: Strategies For Qualitative Research*. Aldine Transaction, 1967.

Robert L. Glass. What's Wrong with Software Reuse?, 2001. URL `http://www.stickyminds.com/s.asp?F=S2731_COL_2`.

Robert L. Glass. *Software Creativity 2.0.* d.* Books, 2nd edition, 2006a.

Robert L. Glass. The Standish Report : Does It Really Describe a Software Crisis ? *Communications of the ACM*, 49(8):15–16, 2006b.

Erving Goffman. Embarrassment and social organization. *American Journal of Sociology*, 62(3):264–271, November 1956.

Erving Goffman. *The presentation of self in everyday life.* Doubleday, 1959.

Erving Goffman. *Interaction Ritual: essays on face-to-face behaviour*, chapter On face-work: an analysis of ritual elements in social interaction. Anchro Books, New York, 1964.

Charles Goodwin. Action and embodiment within situated human interaction. *Journal of Pragmatics*, 32:1489–1522, 2000.

Anandavisam Gopal, Tridas Mukhopadhyay, and Mayuram Krishnan. The Role of Software Processes and Communication in Offshore Software Development. *Communications of the ACM*, 45(4):193–200, April 2002.

Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. The Geography of Coordination : Dealing with Distance in R & D Work. In *Proceedings of GROUP 99*, pages 306–315, 1999.

GWT. Google Web Toolkit. URL `https://developers.google.com/web-toolkit/`.

Mark Handel and James D. Herbsleb. What is chat doing in the workplace? In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 1–10. ACM, 2002.

Jo E. Hannay, Tore Dybå, Erik Arisholm, and Dag I.K. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Information and Software Technology*, 51(7):1110–1122, July 2009.

Jo E Hannay, Erik Arisholm, Harald Engvik, and Dag I K Sjøberg. Effects of Personality on Pair Programming. *IEEE Transactions on Software Engineering*, 36(1):61–80, 2010.

Geir K. Hanssen, Aiko Fallas Yamashita, Reidar Conradi, and Leon Moonen. Maintenance and agile development: Challenges, opportunities and future directions. In *2009 IEEE International Conference on Software Maintenance*, pages 487–490. IEEE, September 2009.

Richard Harper, Christian Bird, Thomas Zimmermann, and Brendan Murphy. Dwelling in software: Aspects of the felt-life of engineers in large software projects. In *ECSCW 2013: Proceedings of the 13th European Conference on Computer Supported Cooperative Work, 21-25 September 2013, Paphos, Cyprus,* pages 163-180. Springer, 2013.

Orit Hazzan, Tali Seger, and Gil Luria. How Did the Originators of the Agile Manifesto Turn from Technology Leaders to Leaders of a Cultural Change?, February 2010. URL http://www.infoq.com/articles/manifesto-originators.

James L. Heap. Conversation analysis methods in researching language and education. In *Encyclopedia of language and education,* pages 217-225. Springer, 1997.

C Heath, H Knoblauch, and P Luff. Technology and social interaction: the emergence of 'workplace studies'. *The British Journal of Sociology,* 51(2): 299-320, June 2000.

Christian Heath and Paul Luff. *Technology in Action.* Cambridge University Press, 2000.

James D. Herbsleb and Audris Mockus. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Transactions on Software Engineering,* 29(6):481^494, June 2003.

James D. Herbsleb, David L. Atkins, David G. Boyer, Mark Handel, and Thomas A. Finholt. Introducing instant messaging and chat in the workplace. In *Proceedings of the SIGCHI conference on Human factors in computing systems,* pages 171-178. ACM, 2002.

John Heritage. Goffman, Garfinkel and conversation analysis. In Margaret Weatherell, Stephanie Taylor, and Simeon Yates, editors, *Discourse Theory and Practice: a reader,* pages 47-56. Sage Publications, first edition, 2001.

John Heritage. Conversation analysis and institutional talk. *Handbook of language and social interaction,* pages 103-147, 2005.

John Heritage. Conversation analysis as social theory. *The new Blackwell companion to social theory,* pages 300-320, 2008.

Morten Hertzum and Annelise Mark Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing and Management,* 36:761-778, 2000.

Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1):75–105, 2004.

Janet Holmes. Modifying Illocutionary Force. *Journal of Pragmatics*, 8:345–365, 1984.

Janet Holmes. When small talk is a big deal: Sociolinguistic challenges in the workplace. *Second language needs analysis*, pages 344–372, 2005.

Janet Holmes. Making Humour Work: Creativity on the Job. *Applied Linguistics*, 28(4):518–537, December 2007.

Janet Holmes and Meredith Marra. Having a laugh at work: how humour contributes to workplace culture. *Journal of Pragmatics*, 34:1683–1710, 2002.

H. Holmström, B. Fitzgerald, P.J. Ågerfalk, and E.Ó. Conchúir. Agile practices reduce distance in global software development. *Information Systems Management*, Summer:7–18, 2006.

Dave H. Hoover and Adewale Oshineye. *Apprenticeship Patterns. Guidance for the aspiring software craftsman*. O'Reilly and Associates, 2010.

Robert J. House. Scientific investigation in management. *Management International Review*, 10(4/5):139–150, 1970.

Hai Huang, WT Tsai, S. Bhattacharya, XP Chen, Y. Wang, and J. Sun. Business rule extraction from legacy code. In *Computer Software and Applications Conference, 1996. COMPSAC'96., Proceedings of 20th International*, pages 162–167. IEEE, 1996.

John Hughes, Jon O'Brien, Tom Rodden, Mark Rouncefield, and Ian Sommerville. Presenting ethnography in the requirements process. In *Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on*, pages 27–34. IEEE, 1995.

Jez Humble and Joanne Molesky. Why enterprises must adopt devops to enable continuous delivery. *Cutter IT Journal*, 24(8):6, 2011.

Ian Hutchby. Beyond Agnosticism?: Conversation Analysis and the Sociological Agenda. *Research on Language & Social Interaction*, 32(1):85–93, 1999.

Dell Hymes. Toward ethnographies of communication. *Language and literacy in social practice: A reader*, page 11, 1994.

Sylvia Ilieva, Penko Ivanov, and Eliza Stefanova. Analyses of an agile methodology implementation. In *Proceedings of the 30th EUROMICRO Conference*. IEEE Computer Society, 2004.

Matthias Jarke, X. Tung Bui, and John M. Carroll. Scenario Management: An Interdisciplinary Approach. *Requirements Engineering*, pages 1–31, 1999.

Gail Jefferson. Is "no" an acknowledgement token? comparing american and british uses of (+)/(-) tokens. *Journal of Pragmatics*, 34(10):1345–1383, 2002.

James J. Jiang, Gary Klein, Hsin-Ginn Hwang, Jack Huang, and Shin-Yuan Hung. An exploration of the relationship between software development process maturity and project performance. *Information & Management*, 41 (3):279–288, 2004.

Jaak Jurison. Software project management: the manager's view. *Communications of the AIS*, 2(3):2, 1999.

Henrik Kniberg. Scrum and XP from the Trenches. Technical report, Crisp, 2007.

Karin D Knorr-Cetina. How Superorganisms Change : Consensus Formation and the Social Ontology of High- Energy Physics Experiments. *Social Studies of Science*, 25(1):119–147, 1995.

Osamu Kobayashi, Mitsuyoshi Kawabata, Makoto Sakai, and Eddy Parkinson. Analysis of the interaction between practices for introducing xp effectively. In *Proceedings of the 28th international conference on Software engineering*, pages 544–550. ACM, 2006.

Julia Kotlarsky and Ilan Oshri. Social ties, knowledge sharing and successful collaboration in globally distributed system development projects. *European Journal of Information Systems*, 14(1):37–48, March 2005.

Robert E. Kraut and Lynne A. Streeter. Co-ordination in software development. *Communications of the ACM*, 38(3):69–81, March 1995.

Robert E. Kraut, Carmen Egido, and Jolene Galegher. Patterns of Contact and Communication Collaboration in Scientific Research Collaboration. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work*. IEEE, 1988.

Pikka-Maaria Laine and Eero Vaara. Struggling over subjectivity: A discursive analysis of strategic development in an engineering group. *Human Relations*, 60(1):29–58, January 2007.

Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003.

Bruno Latour. Visualization and Cognition: thinking with eyes and hands. *Knowledge and Society*, 6(1), 1986.

Bruno Latour. *Science in action*. Harvard University Press, 1987.

Jean Lave and Etienne Wenger. *Situated Learning: legitimate peripheral participation*. University of Cambridge, 1991. ISBN 0 521 42374 0. URL `http://books.google.co.uk/books?id=CAVIOrW3vYAC`.

John Law. Technology and heterogeneous engineering: the case of Portugese expansion. In Wiebe E Bijker, Thomas Hughes, and Trevor J Pinch, editors, *The Social Construction of Technological Systems*, page 405. MIT Press, 1987.

Philip Lawrence and Jim Scanlan. Planning in the dark: why major engineering projects fail to achieve key goals. *Technology Analysis & Strategic Management*, 19(4):509–525, 2007.

Lucas Layman, Laurie Williams, Daniela Damian, and Hynek Bures. Essential communication practices for Extreme Programming in a global software development team. *Information and Software Technology*, 48(9):781–794, March 2006. ISSN 09505849.

María Lázaro and Esperanza Marcos. Research in software engineering: Paradigms and methods. In *CAiSE Workshops (2)*, pages 517–522, 2005.

Mark R Leary and Robin M Kowalski. Impression Management : A Literature Review and Two-Component Model. *Psychological Bulletin*, 107(I):34–47, 1990.

Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation. *IEEE Software*, November, 2003.

Timothy C. Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: data collection techniques for software field studies. *Empirical Software Engineering*, 10:311–341, 2005.

Claude Levi-Strauss. *The Savage Mind*. Weidenfeld and Nicholson, London, 1966.

Stephen Linstead. From postmodern anthropology to deconstructive ethnography. *Human Relations*, 46(1):97–120, 1993.

Nick Llewellyn and Jon Hindmarsh. *Organisation, interaction and practice: Studies of ethnomethodology and conversation analysis*. Cambridge University Press, 2010.

Peter Lloyd. Storytelling and the development of discourse in the engineering design process. *Design Studies*, 21:357–373, 2000.

Mike Loukides. *What is DevOps?* O'Reilly Media, 2012.

Janet Low, Jim Johnson, Pat Hall, Fiona Hovenden, Janet Rachel, Hugh Robinson, and Steve Woolgar. Read this and change the way you feel about software engineering. *Information and software technology*, 38(2):77–87, 1996.

Michael Lynch and Mark Peyrot. Introduction: A Reader's Guide to Ethnomethodology. *Qualitative Sociology*, 15(2):113–122, 1992.

Lech Madeyski. On the effects of pair programming on thoroughness and fault-finding effectiveness of unit tests. In *Product-Focused Software Process Improvement*, pages 207–221. Springer, 2007.

Robert Cecil Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR, 2003.

J. Mason. Mixing methods in a qualitatively driven way. *Qualitative Research*, 6(1):9–25, February 2006.

Pete McBreen. *Software Craftsmanship: the new imperative.* Addison Wesley, September 2001. ISBN 0201733862.

Ian R. McChesney and Séamus Gallagher. Communication and co-ordination practices in software engineering projects. *Information and Software Technology*, 46(7):473–489, June 2004.

Laurie McLeod, Stephen G. MacDonell, and Bill Doolin. Qualitative research on software development: a longitudinal case study methodology. *Empirical Software Engineering*, 16(4):430–459, January 2011.

Allen E. Milewski. Global and task effects in information-seeking among software engineers. *Empirical Software Engineering*, 12(3):311–326, January 2007.

Brian Moeran. *Ethnography at work.* Berg, 2006.

Harvey Molotch. *Where stuff comes from.* Routledge, 2003.

Gregory Moorhead, Christopher P. Neck, and Mindy S. West. The Tendency toward Defective Decision Making within Self-Managing Teams: The Relevance of Groupthink for the 21st Century. *Organisational behaviour and human decision processes*, 73(2/3):327–351, March 1998.

Mario E Moreira. Treating agile as a transformation project. In *Being Agile*, pages 105–112. Springer, 2013.

Matthias M Muller and Frank Padberg. An empirical study about the feelgood factor in pair programming. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 151–158. IEEE Computer Society, 2004.

Kumiyo Nakakoji, Yasuhiro Yamamoto, and Yunwen Ye. Supporting Software Development as Knowledge Community Evolution. In *Proceedings of Supporting the Social Side of Large Scale Software Development*, pages 31–35, Banff, Canada, 2006. Microsoft Research.

Peter Naur and Brian Randell. *Software Engineering: Report on a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, Garmisch, Germany, October 1968.

Jim Q. Ning, Andre Engberts, and W. Voytek Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, 1994.

James Noble and Robert Biddle. Notes on postmodern programming. In *Proceedings of the Onward*, Seattle, Washington, USA, 2002. ACM.

Seán Ó'Riain. *Net-Working for a Living: Irish Software Developers in the Global Workplace*, pages 15–39. Blackwell Publishing Ltd, 2008.

Julian Orr. Ten Years of Talking About Machines. *Organization Studies*, 27(12): 1805–1820, December 2006.

Julian E. Orr. *Talking about machines*. Technology and work. Cornell University Press, first edition, 1996.

James D Palmer and N Ann Fields. Computer-Supported Cooperative Work. *Computer*, 27(5):15–17, 1994.

David Lorge Parnas. Successful software engineering research. *ACM SIGSOFT Software Engineering Notes*, 23(3):64–68, May 1998.

Leslie Perlow and John Weeks. Who's Helping Whom? Layers of Culture and Workplace Behavior. *Journal of organizational behavior*, 23(4):345–361, June 2002.

Leslie A. Perlow. The time famine: Toward a sociology of work time. *Administrative Science Quarterly*, 44(1):57–81, 1999.

Henry Petroski. *To Engineer is Human*. Vintage Books, 1992.

Henry Petroski. *Invention By Design*. Harvard University Press, first edition, 1996.

M. Pikkarainen, O. Salo, R. Kuusela, and P. Abrahamsson. Strengths and barriers behind the successful agile deployment—insights from the three software intensive companies in Finland. *Empirical Software Engineering*, 17: 675–702, 2012.

Barbara Plester and Janet Sayers. "Taking the piss": Functions of banter in the IT industry. *Humor – International Journal of Humor Research*, 20(2):157–187, 2007.

Laura Plonka, Judith Segal, Helen Sharp, and Janet van der Linden. Collaboration in pair programming: driving and switching. In *XP 2011 : 12th International Conference on Agile Software Development*, pages 43–59. Springer, 2011.

Jonathan Potter. *Representing Reality*. Sage Publications, 1996.

Steven R Rakitin. Manifesto elicits cynicism. *IEEE Computer*, page 5, December 2001.

Anne Warfield Rawls. Harold garfinkel, ethnomethodology and workplace studies. *Organization Studies*, 29(5):701–732, 2008.

Linda Rising and Norman S. Janoff. The scrum software development process for small teams. *IEEE Software*, July/August:26–32, August 2000.

Hugh Robinson, Judith Segal, and Helen Sharp. Ethnographically-informed empirical studies of software practice. *Information and Software Technology*, 49:540–551, 2007.

Heather Rolfe. Skill, deskilling and new technology in the non-manual labour process. *New technology, work and Employment*, 1(1):37–49, 1986.

Kari Ronkko. Interpretation, interaction and reality construction in software engineering: An explanatory model. *Information and Software Technology*, 49: 682–693, 2007.

Kari Ronkko, Olle Lindeberg, and Yvonne Dittrich. 'Bad Practice' or 'Bad Methods' Are Software Engineering and Ethnographic Discourses Incompatible? In *International Symposium on Empirical Software Engineering*. IEEE, 2002.

Scott Rosenberg. *Dreaming in code*. Crown Publishers, 2007.

Winston W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*. IEEE, 1970.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14 (2):131–164, 2009.

Ioana Rus and Mikael Lindvall. Knowledge Management in Software Engineering. *IEEE Software*, 2002(May/June):26–38, May 2002.

Steve Sawyer, Joel Farber, and Robert Spillers. Supporting The Social Processes of Software Development. *Information Technology and People*, 10(1): 46–62, 1997.

Ulrike Schultze. A Confessional Account of an Ethnography about Knowledge Work. *MIS Quarterly*, 24(1):3–41, 2007.

Ken Schwaber. Scrum Development Process. In Rebecca Wirfs-Brock, editor, *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 10–19. ACM, 1995.

Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4), August 1999.

Carolyn B. Seaman and Victor R. Basili. Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering*, 24(7):559–572, July 1998.

John R. Searle. Collective Intentions and Actions. In Philip R Cohen, Jerry Morgan, and Martha Pollack, editors, *Intentions in Communication*. MIT Press, 1990.

Peter Seibel. *Coders At Work*. Apress, 2009.

Richard Sennett. *The Craftsman*. Allen Lane, 2008.

Thilagavathi Shanmuganathan. Ethics and the Observer's Paradox. *RELC Journal*, 36(1):73–84, April 2005.

Helen Sharp and Hugh Robinson. An ethnography of XP practice. In *15th Workshop of the Psychology of Programming Interest Group*, volume 44, pages 121–122, April 2003.

Helen Sharp and Hugh Robinson. An ethnographic study of XP practice. *Empirical studies of software engineering*, 9:353–375, 2004.

Helen Sharp and Hugh Robinson. Collaboration in mature XP teams, September 2006.

Helen Sharp, Hugh Robinson, and Mark Woodman. Using Ethnography and Discourse Analysis to Study Software Engineering Practices. In *Twenty-second International conference on software engineering*, pages 81–87, 2000.

Helen Sharp, Mark Woodman, and Fiona Hovenden. Tensions around the adoption and evolution of software quality management systems: a discourse analytic approach. *International Journal of Human-Computer Studies*, 61(61):219–236, August 2004.

Clay Shirkey. *Here Comes Everybody*. Allen Lane, first edition, 2008.

Jonathan Sillito and Eleanor Wynn. Social Dependencies and Contrasts in Software Engineering Practice. In *Proceedings of Supporting the Social Side of Large Scale Software Development*, pages 47–50, Banff, Canada, 2006.

Dag I. K. Sjoberg, Tore Dybå, and Magne Jorgensen. The Future of Empirical Methods in Software Engineering Research The Future of Empirical Methods in Software Engineering Research. In *Future of Software Engineering 2007*. IEEE, 2007.

Victor Skowronski. Do Agile Methods marginalize Problem Solvers? *Computer*, October:118–120, 2004.

Kari Smolander. Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In *Proceedings International Symposium on Empirical Software Engineering*, pages 211–221. IEEE Computer Society, 2002.

Software Engineering Institute. CMMI — Overview, 2010. URL `http://www.sei.cmu.edu/cmmi/`.

Ian Sommerville. *Software Engineering*. Addison Wesley, seventh edition, 2004. ISBN 0321210263.

Ian Sommerville, Tom Rodden, Pete Sawyer, Richard Bentley, and Michael Twidale. Integrating ethnography into the requirements engineering process. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 165–173. IEEE, 1993.

Christoph Steindl and Pal Krogdahl. Estimation in Agile Projects Software is New Product Development. Technical report, IBM Global Services, 2005.

Katherine J Stewart and Sanjay Gosain. THE IMPACT OF IDEOLOGY ON EFFECTIVENESS IN OPEN SOURCE SOFTWARE DEVELOPMENT TEAMS. *MIS Quarterly*, 3224, April 2006.

Maria Stubbe, Chris Lane, Jo Hilder, Elaine Vine, Bernadette Vine, Meredith Marra, Janet Holmes, Ann Weatherall, and J O Hilder. Multiple Discourse Analyses of a Workplace Interaction. *Discourse Studies*, 5(3):351–388, August 2003.

Lucy Suchman. *Plans and situated actions: the problems of human machine communication*. Cambridge University Press, 1987.

Lucy Suchman. Located Accountabilities in Technology Production. Technical report, Centre for Science Studies, Lancaster University, UK, Lancaster, 2003. URL `http://www.comp.lancs.ac.uk/sociology/papers/Suchman-Located-Accountabilities.pdf`.

Lucy Suchman, Randall Trigg, and Jeanette Blomberg. Working artefacts: ethnomethods of the prototype. *The British Journal of Sociology*, 53(2):163–179, 2002.

Lucy A Suchman and Randall H Trigg. Artificial Intelligence as Craftwork. In Seth Chaiklin and Jean Lave, editors, *Understanding practice Perspectives on activity and context*, page 400. University of Cambridge, 1996.

Y Sugimori, K Kusunoki, F Cho, and S Uchikawa. Toyota production system and Kanban system Materialization of just-in-time and respect-for-human system. *Internation Journal of Production Research*, 15(6):553–564, 1977.

Jeff Sutherland and Ken Schwaber. The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process. Technical report, The Scrum Foundation, 2007. URL `http://assets.scrumfoundation.com/downloads/2/scrumpapers.pdf?1285932052`.

Jeff Sutherland, Anton Viktorov, and Jack Blount. Distributed Scrum: Agile Project Management with Outsourced Development Teams. In *The Proceedings of Agile 2006*, 2006.

Stephanie Teasley, Lisa Covi, M. S. Krishnan, and Judith S. Olson. How Does Radical Collocation Help a Team Succeed? In *CSCW '00*, pages 339–346, Philadeplhia, USA, 2000. ACM.

Paul Ten Have. *Doing conversation analysis*. Sage, 2007.

Stella Ting-Toomey, editor. *The Challenge of Facework: cross-cultural and interpersonal issues*. State University of New York Press, Albany, New York, USA, 1 edition, 1994.

Gerard a Tobin and Cecily M Begley. Methodological rigour within a qualitative framework. *Journal of advanced nursing*, 48(4):388–96, November 2004.

Franziska Tschan. Communication Enhances Small Group Performance if it Conforms to Task Requirements: The Concept of Ideal Communication Cycles. *Basic and Applied Social Psychology*, 17(3):371–393, 1995.

Dan Turk and Robert France. Limitations of agile software processes. In *Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2002.

Michael L. Tushman. Technical Communication in R & D Laboratories : The Impact of Project Work. *The Academy of Management Journal*, 21(4):624–645, 1978.

Michael Twidale, Tom Rodden, and Ian Sommerville. The Designers Notepad: Supporting and understanding cooperative design. In *Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, pages 93–108, Milan, Italy, 1993. Kluwer Academic Press.

John Van Maanen. *Tales of the field: On writing ethnography*. University of Chicago Press, 2011.

K. E. Weick and K. H. Roberts. Collective mind in organizations: Heedful interrelating on flight decks. *Administrative science quarterly*, pages 357–381, 1993.

Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings of the 17th international conference on Software engineering*, pages 327–331. ACM, 1995.

Gerald M. Weinberg. *The Psychology of Computer Programming*. Dorset House, silver ann edition, 1998.

Gerald M. Weinberg. Egoless programming. *IEEE Software*, 16(1):118–120, 1999.

Etienne Wenger. Communities of practice: Learning as a social system. *Systems thinker*, 9(5):2–3, 1998.

Etienne Wenger. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press, 1999. ISBN 0 521 66363 6. URL `http://books.google.co.uk/books?id=heBZpgYUKdAC`.

Etienne C. Wenger and William M. Snyder. Communities of practice: The organizational frontier. *Harvard business review*, 78(1):139–146, 2000.

Elizabeth Whitworth and Robert Biddle. The Social Nature of Agile Teams. In *Agile 2007 (Agile 2007)*, pages 26–36. Ieee, August 2007.

Laurie A. Williams and Robert R. Kessler. All i really need to know about pair programming i learned in kindergarten. *Communications of the ACM*, 43(5): 108–114, 2000.

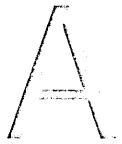
Ruth Wodak. What cda is about – a summary of its history, important concepts and its developments. In Ruth Wodak and Michael Meyer, editors, *Methods of Critical Discourse Analysis*, pages 1–13. Sage Publications, 2001.

Rebecca Yates. Conducting field studies in software engineering: an experience report. In *Proceedings of PPIG 2012*, 2012.

Yunwen Ye, Yasuhiro Yamamoto, and Kouichi Kishida. Dynamic Community: A New Conceptual Framework for Supporting Knowledge Collaboration in Software Development. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2004.

Ed Yourdon. *Death March*. Prentice Hall, second edition, 2003. ISBN 013143635X.

Symbols used in the transcriptions



[	The point at which overlapping speech begins.
]	The point at which overlapping speech ends.
=	There is no break in speech. Pairs of equals signs indicate no break between two lines.
(0.0)	A noticable pause within a segment of talk which can be measured in seconds.
(.)	A brief pause within a segment of talk.
:	An elongation of a word or part of a word. Single colons indicate a brief lengthening, pairs of colons indicate that the word is stretched over a longer period.
{...}	Non-vocalisations such as coughs, laughs etc.
> <	Speech slows down.
< >	Speech speeds up.
↑	Rising intonation.
°	Soft sounds.
'	Part of the word is not pronounced.

Advertising the research B

This Appendix includes the flyer and Webpages through which the project was advertised. The Website can be found at `http://homepages.shu.ac.uk/~cmscb/scrum`.

DISCOVERING WHAT REALLY HAPPENS INSIDE AN AGILE TEAM

Introduction

Background

The fieldwork

How to help

Sharing the findings

Safeguards

Contact details

My name is Chris Bates. I'm a Senior Lecturer in Software Engineering at Sheffield Hallam University. These Web pages describe a research project which I am undertaking to examine what *really* happens inside teams of developers when they adopt agile methods.

This site constitutes a plea for help. I'm looking for a development team who wouldn't mind having an academic researcher come along and watch them at work. If you are working in an agile team, in particular one using Scrum please get in touch using these details.

You may have questions and comments when you've looked around this site. Please feel free to email me if you want to discuss this work.

DISCOVERING WHAT REALLY HAPPENS INSIDE AN AGILE TEAM

Introduction
Background
The fieldwork
How to help
Sharing the findings
Safeguards
Contact details

Hopefully whether you've seen one of my flyers, heard me ask for help somewhere or read the introduction page you are intrigued about the project. In the section on fieldwork you will read about what I'm intending to do.

Why go and look at what happens inside a scrum instead of using the method and finding if it works for you?

There is a stereotype of the software engineers as a lone coder hacking away in a darkened room or cubicle surrounded by thick programming books and Star Wars or Red Dwarf memorabilia. In the popular imagination these engineers aren't interested in the users of their software and tend not to work well in teams. I'm not sure that this cliche has any validity today, and I'm not really sure that it ever did. The development of software is a communal activity which involve teams of designers, testers, coders and users.

Agile methods are predicated on the social aspects of software development. All of them base their practices on people working closely together rather than going off and doing their own thing. Think about the XP practice of pair programming. If two developers share a single keyboard and screen then they must talk about the code which they are writing. On-site customers are there to talk to the developers. And methods such as scrum are built on regular meetings within the team. You can't be an agile developer without also being a social developer.

Introduction
Background
The fieldwork
How to help
Sharing the findings
Safeguards
Contact details

I'm looking for an enthusiastic team who are experienced with scrum and who are located within an hour of Sheffield.

If you would like to get involved or simply want to know more please contact me. You may not be in a position to invite me along to your project but you will know the person who *is*. Clearly there is a going to have to be negotiation before I come in. I'll do that, working with both developers and managers to create a process which works for us all. All you have to do is to let me know who to talk to and tell them that you've been in touch with me. it would help if you could also tell me a little about your projects.

I hope to do my fieldwork during late Winter and early Spring 2010.

Are your projects agile?
Do you use scrum?

Could you help me with my research?

What is the research?

_m an academic researcher and experienced software developer
:amining the scrum methodology. I'm not interested in scrum as an
-proach to project management. I am interested in how developers use
■rum to change their approach to software development

Key questions

want to know what really happens during scrums. Does the method
lange the ways in which programmers work together? Are these changes
meficial?

hy does scrum appeal to so many programmers - and why does it appal
:hers.

er wonder why you have to battle to be agile in the face of RUP or
"ince2?

Become involved

you are involved in a scrum as Product Owner, Scrum Master or Team
ember I'd like to talk to you

your organisation uses scrum I'd like to come and watch you at work

Contact

Chris Bates
Senior Lecturer in Software Engineering
Email: c.d.bates@shu.ac.uk
Phone: 0114 274 8015
Skype: floydheel

anemeia
‘ Hallam University
HARPENS YOUR THINKING

Chickens and pigs

A pig and a chicken are walking down a road. The chicken looks at the pig and says, "Hey, why don't we open a restaurant?" The pig looks back at the chicken and says, "Good idea, what do you want to call it?" The chicken thinks about it and says, "Why don't we call it 'Ham and Eggs'?" "I don't think so," says the pig, "I'd be committed, but you'd only be involved."

Consent form C

This appendix is the consent form which was offered to participants in the study.

Sheffield.
Hallam
University

PARTICIPANT CONSENT FORM

TITLE OF RESEARCH STUDY:

Please answer thefollowing questions by ticking the response that applies

		YES
1.	I have details of the study explained to me.	I i
2.	My questions about the study have been answered to my satisfaction and I understand that I may ask further questions at any point.	□
3.	lunderstand that 1 am free to withdraw from the study within one week of the completion of this fieldwork, without giving a reason for my withdrawal or to decline to answer any particular questions in the study without any consequences to my future treatment by the researcher.	□
4.	I agree to provide information to the researchers under the conditions of confidentiality set out in the briefing.	I I
5.	1 wish to participate in the study under the conditions set out in the briefing.	□
6.	**I** consent to the information collected for the purposes of this research study, once anonymised (so that I cannot be identified), to be used for any other research purposes.	

Participant's Signature:____

Date:_____

Participant's Name (Printed):

Contact details:

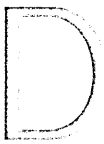
Researcher's Name (Printed):_____

Researcher's Signature:_____

Researcher's contact details:
(Name, address, contact number of investigator)

Please keep your copy of the consent form and the information sheet together.

The daily stand-up at A*



1	Dan	...because you're rubbish Gary
2	Gary	oh he's in he's in
3	Dan	hooray
4	Gary	yay
5	Dan	is he a silent partner
6	Gary	he is he is he's nothing laughs
7	Dan	so we can say whatever we like about Chris
8	Gary	Ideally he doesn't want us to be all professional pretend like we're a real company he want us to you know laugh
9	All	*laughter*
10	Gary	Call each other cocks like we usually do
11	All	*laughter*
12	Dan	la la la ok then (0.7) errm did Evan ever show up (.)
13		[no]
14	Ed	[doe]sn't look like it (.) no
15	Gary	[no]
16	Dan	good work Kippers
17		right then let's errm go round the table (.) let's start with Gaz ↑

18	Gary	yay (.) so err yesterday I err was implementing the service for query objects and did that so that's finished now errm and then I moved on to implementin the queries on the field rules and I've done all of them apart from the partial date field rule which I was just lookin how to do now 'cause obviously I'm goin to need to build up the query for that which is going to be a bit different to just a date query (0.5)
19		So I figures out how to do it (.) Dan I think I was talking to you yesterday about that
20	Dan	Ah ha
21	Gary	So I can use the to car char and just pass in a date sniffs and a month so I'm going to have to write a criteria a match criteria object for that an' that should work [background pinging noise]
22		And so that's what I'm gonna be doin this morning and then this afternoon just carry on implementing
23		the err [query obj]ect
24	Dan	[did you] ever get it to a big sort of data set of millions or did you give up
25	Gary	Errm I got it to two and a half million
26	Dan	That's files
27	Gary	Yeah (0.7) or maybe slightly more than that and then (1.2)
28	Dan	Or did you ever figure out why it was slow on your computer
29	Gary	No I didn't
30	Dan	(2.0) hummm
31	Gary	I mentioned it Evan and he said well if it's gonna work in the short term [we're not]
32	Dan	[yeah] there is that I suppose
33	Gary	we haven't got millions now↑ he says just get it done now and then obviously later on when we do get loads more (1.0) users and files we can we can try it
34	Dan	We can JIT it

35	Gary	We can do something else just too late I mean just in time yeah
36	Dan	Yeah [laughs]
37	Gary	[laughs]
38	Dan	Okey dokey Ed
39	Ed	OK yeah I was fiddling around springificating all the insurance stuff
40		Errm (1.8) so yeah spent a lot of time annotating classes and turning that thing into a contribution (.) the err primitive mapping stuff
41		Errm (2.1) in the interludes I started to think about this service for kind of err cloning the database errm (1.6) so writing some notes and started a bit on that
42	Dan	Interludes
43	Ed	Well [like]
44	Dan	[you b]een watching a show or something
45		*General laughter*
46	Ed	No no no the bits where I gave it back to you and you were faffing around with [it]
47	Dan	[laughs]
48	Gary	Is that our word of the day↑
49	Ed	Errm so at the moment errm (1.5) bit of a weird one when you attempt to go into the services it doesn't actually ex exception (0.9) but it hangs indefinitely↑ errm so I'm trying to figure out [what's going on]
50	Dan	[I was I] was getting that(0.5)
51	Ed	Right
52	Dan	With some of the stuff I've done errm (0.4) when I was trying to reference a spring bean from tapestry↑
53	Ed	Right
54	Dan	But I hadn't given the spring bean explicitly a name↑
55	Ed	Right (.) ok
56	Dan	And it was just hanging↑
57	Ed	Right ok

58	Dan	and I couldn't work out why until I gave it a name and it worked
59	Ed	OK
60	Dan	So maybe have a look at that↑
61	Ed	Right yeah I'll give that a shot then Err I was thinking it was something to do with Hibern↑ate to be honest 'cause that's the last thing you see
62	Dan	Yeah it's opening a hibernate session m[ine wa]s
63	Ed	[yeah]
64	Dan	And then it just it was obviously somewhere trying to get to a (0.8) bean and failing but I don't know why it hangs it's a bit random↑
65	Ed	OK so yeah figuring that one out really errm
66	Dan	It'll be tha' it'll be fixed in no time
67	Ed	*Laughs*
68	Dan	(1.6) So what else you doin'
69	Ed	Errm so well I mean however long that takes and yeah moving on [to]
70	Dan	[Are] you close 'cause you committed a bunch of stuff last nigh didn't you
71	Ed	(0.2) Well I mean that was enough stuff (.) that you know it no longer exceptioned (0.2) well I thought it was gonna work and then it just ended up hanging like this so (1.4)
72	Dan	*Grunts*
73	Ed	But the I thought sod it you know it was forty files (.) I don't wanna leave 'em hangin' around too long so (0.2) checked it all in (1.2) break stuff a bit more *laughs*
74	Dan	You know the original bug you were fixing with utility files and stuff
75	Ed	Yep
76	Dan	That is fixed isn't it
77	Ed	Yeah yeah I saw Evan asked whether I don't I just don't think it made it on to demo 'cause other stuff got committed and
78	Dan	[Yeah that's]

79	Ed	[I don't] think it was stable to enough to put another build up on there so
80	Dan	That's what I thought (1.8)
81		Ok well I I I just you need you to sort of (0.5) finish that so (0.4) we can get your finger out of the springy stuff and I can start putting my finger into it
82	Ed	*Laughs* [Right]
83	Dan	[I don't] wanna start modifying 'em all if you're still
84	Ed	Ok yep sure *laughs*
85	Dan	(2.0) And then you're going on to (0.2) stuff with Gaz yeah
86	Ed	With Gaz yeah yeah
87	Dan	(1.6) >Fine< (0.4) <Ok> I'll add what I was doing as a developer
88		I err (1.8) got a file >session< to work in spring that tapestry could see which was for Ed (0.3) and then I was getting the changes that I did in Sheffield off my laptop onto my other one which was getting the stylesheet stuff to work
89		and that's when I cam across the same issue (.) Ed had with the err it just hanging (0.3) so I worked that one out
90		(1.0) Errm and I (.) I started doing the bits on the log in pages to make it exactly the same as before (0.4) so that's what I'll finish first this morning until I can (0.4) get on with the rest of spring
91		(1.3) Errm (2.0) >so that's me (.) let's see what < Will did (1.1)
92	Will	So err yes'day I started of helping Sam with his waters err (.) didn't really take very long doing that 'cause he's getting (0.5) fairly confident at doing them himself [now]
93	Sam	[Too right]

94	Will	That's useful (.) I can go and get on with my own stuff now (0.5) errm (2.2) so yeah started looking at my own (.) water tests (.) err fixed a couple of those (1.1) err not sure wever the locking thing with Frankie↑ (1.0) is actually (.) a problem with the locking itself or just whether Frankie's being a bit weird at the moment (0.2) because when I went to look at it this morning to see if it had run a' >all< (0.5) it was just sort of there was just nothing there what so ever (0.4) so I tried running it's scheduled task again and it just locked like straight away*uparrow* (1.0) so I'm sure wever it's a locking issue or wever Frankie's just a bit (1.1) [rubbish]
95	Dan	[It used] to work for a while
96	Ed?	Well Frankie's been rubbish
97	Will	It did used to work yeah*uparrow*
98	Dan	Did you guys have a look at that locking issue then did you try and turn it off (0.4)
99	Will	[No we haven]'t tried to no (2.2)
100	Dan	[??? missed]
101	Will	(1.7) Errm so (0.7) yeah today I'm gonna be looking at (.) Ben committed a load of stuff to do with the sign-up page yesterday (.) as well as the reset account page (.) and I fink he mentioned in passing he might (.) or might not have done the activation page (0.3) so I'll have a look at that one as well
102	Dan	He might or might not *laughs*
103	Will	Yeah maybe or maybe not who knows
104	Dan	That's good
105	Sam ??	As committal as ever
106	Will ??	[yeah]
107	Dan	[I] can tell you if he has I don't think he did (0.5)
108	Will	I got the impression he didn't but he thought maybe (.)
109	Dan	Err pages (0.2) sign up (0.2) reset (0.2) there's no activation page in there

110	Will	Ok so I won' I won't even bover looking the activation page then
111	Dan	(1.0) No I think↑ just check that it still works because err (0.2) that's the only page left on public that's tapestry driven (0.4)
112	Will	Oh is it really (.) OK (0.5)
113	Dan	[shou should be]
114	Will	[I'll just check] it's still working now
115	Dan	(1.2) >OK cool< (.) Sam
116	Sam	(1.1) Errm (0.4) yeyesterday I was working on one water test pretty much all day which was the (.) errm complete car insurance quote one (0.2) which is (.) slowly gettin' there (0.5) errm (0.8) had to make lots↑ of interesting definitions and things (1.0)
117		I'm sort of getting this programming thing (.) sorted out now I think (0.6)
118		Errm (0.9) so I'm going to be finishing that off today pretty much
119		(0.4) Errm (0.3) I probably should mention now errm I'm not gonna be here on Monday (1.8) 'cause I [boo I t]hink I booked
120	Dan	[hoorah]
121	Ssm	Yeah I booked it off ages ago but then Ben didn't make a note of it
122	??	Oh Ben
123	Ssm	But he says it's fine↑
124	Dan	(1.7) Ok yeah that's useful to know
125	Sam	Yeah 'cause I'm gonna be hung over (0.7) 'cause it's my birthday
126	Dan	[Oh↑ >exciting<]
127	Sam	[on Sunday] and you're all coming
128	Gary	No we're not (.) honestly
129		*General laughter*
130	Sam	Why is the fuel bill fuel bill gonna be too high in your car Gary
131		(2.2)
132	Gary	What↑
133	Sam	Eighty to the gallon it'll do (1.6)
134	Dan	He only lives down the road from you

135	Gary	Yeah why would I drive
136	Dan	Just roll
137	Sam	It's in Derbyshire it's back home [in the motherlands]
138	Dan	[You're giving everyone] an awesome excuse not to go if it's in Derbyshire
139		*laughter*
140	Will?	(2.3) Stay away
141	Dan	Right what you doin' today (0.2) before you err (0.2) jibber jabbered about your birfday
142	Sam	Err I'm finishin' off this (0.4) test
143	Dan	What you going to do <after> that one
144	Sam	(0.4) Err
145	Dan	Are you going to do like utilities and all of those or have they already got them
146	Sam	(0.2) *Blows out* they're broken
147	Dan	(1.1) They're broken↑
148	Sam	<Yep> (1.4) err utilities is broken on live at the moment or was (0.8) two days ago↑ last time I looked at it
149	Dan	Well it's not going to fix itself (0.5) err are mobile phones in or
150	Sam	Oh I do need to yeah have a look for stuff like that but I need to do (0.4) errm some autocompl>ete< stuff (1.0) for the autocomplete fields (.) so that it can get the value out of it (2.0) which is going to be fun (.) because they don't have a UID
151	Dan	(3.0) Fair enough I'm sure↑ you can work it out (1.4) err
152	Sam	I'll get Will to help me work it out
153	Dan	(1.3) Ok no Evan still (.) >what a< douchebag
154	Gary	We should take a minute to mock where Ben is
155	Ed	(1.1) Oh yeah yeah
156	Sam	Where is he
157	Dan	Ok let's have an update from Ben who's not here (.) he's at a Magic competition
158	Gary	He's at Magic The Gathering
159		*laughter*

160	Dan	And there we go (.) errm
161	Sam	With his special t-shirt on (1.2)
162	Gary	He needs an intervention that boy
163		*laughter*
164	Dan	Has he got a magic t-shirt
165	Gary	Yeah he's got magic t-shirt they got team t-shirts
166	Dan	Hasn't Will got one (0.6)
167	Will	Hey leave me out of this
168		*laughter*
169	Gary	Will's just sitting there doing his work
170	Dan	Have you got one though Will that's the question
171	Will	he did get me one yeah
172	Dan	yeah
173	Gary	Are you gonna wear it though (1.0)
174	Will	[Maybe]
175	Dan	[I bet] he's wearing it now (.) secretly (0.2) hoping the team's gonna win
176	Sam	(0.3) He's gonna wear it to work tomorrow
177	Dan	(1.4) Is there anything else anyone wants to add (0.5) other than Ben at magic
178	Sam	(0.3) There probably was several things that I meant to say and I've forgotten (.) oh who wants tea and who wants coffee↑
179	Gary	(0.3) We'll sort that out after
180	Dan	???? of a scrum call
181	Gary	Unless you want to go and give Dan his tea *laughter*
182	Dan	(2.4) Thank you dear Ok
183	Gary	Job done
184	Dan	Job done go go code team

Working with existing code at E*

1	Darren	This is kind of the start of a process and over time it might evolve (1.0) or it might not
2	Andy	ha ha
3	Darren	tends to depend how things are used.
4	Darren	Errm (1.8)
5	Darren	yes I think for the implementation side of it the first pass our implementation is about 'cause what we kinda doin' is a provider that's got its own little database here
		(1.5)
6	Darren	tha' that's isolated from that so it can store (.) contacts these'll be probably creatin' it's probably a session-type table
		(1.3)
7	Darren	a user-type table (0.8) ah (1.4) and then the data store
		(1.5)
8	Andy	what's the sense around using echo?
		(1.7)
9	Darren	at the moment that functionality's all kind of been turned off 'cause it it got to a certain point (1.1) and it wasn't really
		(2.8)
10	Darren	to [improve the user experience]
11	Andy	[what was the point of th]at?

12	Darren	the (.) the pla' when we did it to echo (.) then that was (.) the database was here the synchronisation was going and the contacts were pushed up
		(1.6)
13	Darren	into this database. THERE WAS various issues (.) and from a usability point of view it wasn't
		(1.0)
14	Andy	Hmm mmm
		(1)
15	Darren	err >working that< well and also echo (0.9) echo was kind of err
		(1.7)
16	Darren	Yeah (1.0) it had problems (0.3) it was kind of a big
17	Andy	laughs
18	Darren	laughs
19		a big test (.) but this is you know this is kinda tryin' a make it easier. I think as well with that echo side of i' when that (1.1) echo was over here
		(2)
20	Darren	and it was also handling the external contacts so it would kinda get them from salesfo:rc:e so you had to go through >the echo interface to get your contacts from salesforce< whereas we're kind of saying that (0.8) you know (1.5) keeping echo more simple for we're exposing it through the esendex API so anyone who consumes this API (.) has then got access to
21	Darren	you know (1.5) keeping echo more simple for we're exposing it through the esendex API so anyone who consumes this API (.) has then got access to (1.3) contacts from different ↑sources
		(7.1)
22	Darren	but (0.6) yeah so at the moment it's not really visible in echo.= =I think you ↑ CAN?
		(1.3)

23	Darren	through the A B test functionality probably turn it on for yourself (.) and find stuff out and sync your phone (0.7) and you've then got contacts (.) <in this database>
		(2.0)
24	Darren	hem
		(1.4)
25	Darren	so sync data sou:r::ce
26	Andy	in theory we could have (.) replicateContacts in there you could have the same contacts in there as in there as in there
		(2.5)
27	Darren	at the moment (.) yeah
		(1.7)
28	Darren	but that sort of functionality's not really publicly available
		(1.8)
29	Darren	for sync'ing with <the phone>(0.7) hem
		(4.0)
30	Darren	and again it it was (.) when this did an upda'e that kind of deleted those contacts and re-created them
		(4.2)
31	Darren	And that was probably done before we did the (.) sort of contact groups so it was less of an issue tha' (0.6) we were <creatin' brand new <contacts every> time?>
		(5.8)
32	Darren	errm
		(2.4)
33	Darren	>so this interface here is quite simple< (.) this (0.7) passes in a session
		(2.0)
34	Darren	>so once authentication's done then ??? SyncML server when it's discussing with the state box it's always passing a ↑ SESSION out (0.6) I think a session just has
		(3.5)
35	Darren	It's got a few things in there (.) it's got a few bits of space

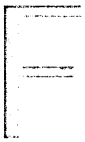
		(14.0)
36	Darren	Ahh?
		(2.5)
37	Darren	Yeah, so the sesh (.) the session object in here this is remembering the whole sort of synchronisation between the phone and server so when I initiate one there's several calls that are made
		(0.9)
38	Darren	I've got a hundred contacts there might be (0.5) you know TEN OR so requests going backwards and forwards (0.6) and this session is (0.5) just for the life of that synchronisation session
		(1.6)
39	Darren	And this is the thing that's holding state betwe:en
40	Darren	(1.4)Errrm(1.0)
41	Darren	Requests so our data structure needs ter (.) store this in its database and if this was with an XML serialisable so that
		(1.3)
42	Darren	You could just put it as a blob (0.5)=
43	Andy	yeah
44	Darren	=in the database or we could break these down to fields (1.0) if we need it
		(4.2)
45	Darren	?? add contacts >there's a session< so un under this implementation there we'd probably just get the session ID
		(1.1)
46	Darren	And there's a load of vcards here (.) these are the new contacts to add
		(4.5)
47	Darren	Errr empty contacts card
		(22)
48	Darren	Ah (.) see this is this part of the err
		(1.9)
49		Stuff where we force it into this slow synch
50	Andy	yeah

51	Darren	So this is where we're actually(1.0) you know clearing out your contacts store so when a new session
		(1.2)
52	Darren	It's it's kind of emptied that (0.8) we need the term contact sto:re was kind of like an earlier (.) grouping idea the contact store these being all the (0.2) all the contacts for this particular ↑ pho:ne
		(1.4)
53	Darren	And it cre I think it did then create (.) a contact store-or a group (0.2) errm (0.2) that was somehow ↑ associated with the phone?
54		(1.0) errm (1.1)
55	Darren	Through the syncML protocol it actu<ally sends up the IMEI ↑ > so we've got quite a unique thing (.) though that is actually to the sim card (0.3) but some fairly
		(1.0)
56	Darren	Once we've (.) created one of these groups that did a new sync again we could at least put the contacts into the same same sort of <grouping>
57	Darren	(1.6) mechanism
		(4.0)
58	Darren	And they contact status
		(3.0)
59	Darren	contact status reference code not sure what that? one
		(12.5)
60	Darren	errm
		(20)
61	Darren	Code returns
		(17.5)
62	Darren	Ah:hh
		(4.6)
63	Darren	Yeah? Of course
		(3.8)

64	Darren	With the (2.0) sort of (.) mo2 project although we did we did synchronisation in one way you could co you could either pu' all your contacts on the phone or take all your contacts off the phone (4.0) so it's kind of an all or nothink
65	Andy	mmmm
66	Darren	Thing (0.5) so (.) so that the (.) the whole premise between the mo2 things are (2.0) important if you've got a new phone how do you get your contacts off the phone so it's a service to do that
67	Andy	Yeah
68	Darren	So here's your old phone (1.0) sync those up to the mo2 service then the new phone sync them back down again as a way of sort of transferring the contacts
69	Darren	(3.0) *coughs* (1.0)
70	Darren	So what >this< is doing here is is that whe when I send send A CONTACT to the phone it's all based on broken down into commands so I'm sending 'em that command (1.0) the phone then (1.0) will respond with some kind of STATUS and it's usually like a http-type (1.0) status code so it's either added it or it's failed to add it it might fail to add it because the err I don't know the contact's too big or or something because we're going from one phone to an↑ other (1.0) this when it it gets a bit [compli]
71	Andy	[Yeah]
72	Darren	cated so (2.0) errm (2.0)
73	Darren	We always deal in the vcard structure and virtual phones (.) the raw data is is the vcard structure but then >a< vcard >can< (1.0) on like an iPhone have an image a >big< image=
74	Andy	Yeah

75	Darren	=of myself and I might transfer that to like a (.) smaller phone and it might nor understand what the image is or have a size limitation (2.0)=
76	Andy	Yeah
77	Darren	=On that kind of thing so there's all (.) all sorts of reasons why adding the contact could fail >but< this is then updating the status back into the data source and that provided a view of saying like
		(1.0)
78	Darren	>I've uploaded< a load of contacts here's my hundred contacts I I've then downloaded to the phone here's the status have they all been successful or have any of them failed (.) we could then find oh actually we sync'd most of them but yer phone didn't like >these< particular contacts
79	Andy	Yeah
80	Darren	(4.0) that kind of thing
		(9.0)
81	Darren	I think
		(4.0)
82	Darren	If we just start off say creatin' a new provider 'cause that's then quite (2.0) quite isolated as a set of interfaces (1.0) we could almost do that
83	Darren	(3.0) errm (3.0)
84	Darren	without actually getting involved in in the ↑ big server that might be a (.) a good way to start so (.) so we create a
85	Andy	Yep
86	Darren	Use a ??? pattern there where we've got a provider(1.0)

Talking about testing at E*



1	Darren	There's probably there's probably a couple of tests we could here one
2	Darren	err from that premise of that (1.0) a the user might not exist (1.0) so one is (1.0) the credentials are always going to be valid we we kind of (1.0) implement from the perspective that I don't care who who the user is (.) if the user is one I don't recognise I'll create a new one otherwise I'll
		(4.0)
3	Andy	So if ??? do log in with invalid credentials to start off wiv
4	Darren	Well that's the thing I'm saying it's never [going] to=
5	Andy	[Yeah]
6	Darren	=be invalid it will be either one of one that exists or one that doesn't so the scenario the two scenarios as I see it that
		(2.5)
7	Darren	Log in with a (1.0) for the first time as a user and then the test=
		(1.5)
8	Darren	=Is probably going to show something like (.) ah try to get hold to get hold of this user from the database the data says it doesn't exist therefore creates a new user in the database and then creates (1.0) a session and returns :that I::d

		(1.5)
9	Darren	And then the other scenario's gonna be where a user sort of doesn't exist (1.0) it's gonna call on to the database
10	Darren	and it's actually O?? gets the user the user does all of these
11	Andy	Have you got your diagram (.) are we what (1.0) we're not even caring
12	Darren	We're not we;re not this bit here we're from the phone
13	Darren	From the phone's perspective but the way that the SyncML is implem:ent:ed=
14	Andy	???
15	Darren	=this this doesn't have any internal (.) authorisation so it it delegates it off to the pro:vider to do (.) hmmm (.) but in this case here
		(2.0)
16	Darren	As an interim thing what I'm saying is that the the
		(3.0)
17	Darren	It's a simple user model in that if err IF YOU log in and they don't recognise it it creates that as a new us er
		(2.0)
18	Andy	When do they actually log in then (.) on the phone
19	Darren	When the the pho when the phone (0.5) so when you set up the ph[one]=
20	Andy	[Yep]
21	Darren	=on your phone you'll have some SyncML settings where you'll detail the name of the serv er (.) the user name password [which]=
22	Andy	[OK]
23	Darren	=You're going to send to it (.) so then on the phone when I (.) go to (.) my synchronisation app and go sync (1.0)
24	Andy	Yep

25	Darren	That first request into <that server> >ACTUALLY that first request might not have< the user name password there might be a (.) handshake where it goes sorry I need a user name password and the phone will send a user name password
26	Andy	Yep
27	Darren	So at that point it then calls on to the
		(4.0)
28	Darren	do you want me to write the test
29	Andy	No I'm just thinking
		(2.0)
30	Andy	So if (1.0) two different mobile phones same user name (.) what happens then
		(3.0)
31	Darren	They're attached to the same u:ser
		(1.5)
32	Darren	What happens what happens then so so in our (.) in our system like with the echo provider that is fine because the >user name password< we're usin' it in Esendex
		(0.5)
33	Darren	And you could have up to five phones so it'll sync with but when it did the sync part of the device information from that pho:ne (.) sends up the IMEI code some kind of unique indicator for that phone or whatever variant of a sim card
		(1.0)
34	Darren	So then it knows that THIS user is talking about THIS phone and these contacts on this phone >so you could then< do a sync on your other pho:nes
45	Darren	So it's not it's not it's not (1.0) necessar (.) it's the same user but that user could have (.) multiple phones it could have multiple groupings of contacts
36	Andy	OK
		(4.0)

37	Andy	So at the moment we:re here's no such thing as valid credentials then
38	Darren	Yeah that's what I'm saying is that is that the only two two scenarios are [logged]=
39	Andy	[logged]
40	Darren	=in with no cre[dent]ials=
41	Andy	[yeah]
42	Darren	=Or known credentials if it's unknown it goes through a process of creating them (.) if they're known (.) it doesn't
		(1.0)
43	Andy	Which one shall we start with
44		(3.0)
45	Darren	I just say I don't know [it's either its]=
46	Andy	[No no I don't]
47	Darren	=going to be the same ??? name (1.0) do a name cos its natural
48	Andy	hnnn
		(3.5)
49	Andy	This keyboard it would be (.) it hurts your wrists
50	Darren	Do you wanna put it down
51	Andy	Ha Do you think we should (1.0) thank you
		(2.5)
52	Andy	I'm gonna make gonna make credent::ials (.) retur::n null what's it return from
53	Darren	It returns that's the session ID
		(5.0)
54	Andy	So this is gonna return null hnnn
55	Darren	No I think we are gonna (.) we are gonna
56	Andy	Oh no (.) no we're not it's gonna return the session ID
		(8.0
57	Andy	It's the log in (1.0) that worked (.) retur::ns
58	Darren	But I think it's also prior to the session that it's going to create create the user (.) in the database
		(20.0)
59	Andy	Put the user

60	Darren	In the session the session
61	Andy	It's a big if
62	Darren	Not too bad I've created bigger
63	Andy	Yes so we're just going to handle this off to somewhere else (2.0) the first one the same

Implementation or design at E*

1	Darren	It's just a simple (.) Do you want me to save the session and err obliterate the database and put it there
2	Andy	OK
		(1.5)
3	Darren	So (.) so that we [FOE1?]have actually now implemented (.) all of the authentication server so so [FOE1?]my concern that this did <???> was in fact that add session was just literally taking an ID from the user ID but then later we're making calls to the repository to get out some XML based on that session? (.) because this decision making process of actually (.) the sync authentication object is not releasing control it's it's deciding I'm going to generate the IDs for the new users=
4	Andy	Yep
5	Darren	=And the new sessions it should also generate a brand new session session=
6	Andy	Yes
7	Darren	=(.) so I'm gonna (.) gonna go back to one of the other tests and change (1.5) so I'm actu actually expecting add session
		(5.0)
8	Andy	I think that the err still still that decision about having these IDs here
		(2.5)

9	Andy	[So now that's doing so much stuff in't it]
10	Darren	[At this stage it's still it's still] yeah bugging me that
11	Andy	Just go back to the code a minute
		(3.0)
12	Darren	We almost need a do we need a mini service there (.) something between this and the errr
		(3.0)
13	Andy	Ummm
		(5.0)
14	Darren	Where this layer (.) actually perhaps this isn't talking to the repository it's talking to a (4.0) service layer and that service layer it's got high level things and that service layer that's responsible (3.0) for creating the user (1.0) ID it's responsible for creating the session
		(2.5)
15	Andy	You're still going to have the same problem though aren't ya (3.0) the same that same you know
		(1.0)
16	Darren	You'll still yeah you'll still get the same problem as when you go down to the [session]
17	Andy	[You're j]ust going to you're just going to copy that method and put it into a (1.0) server side one
		(10.0)
18	Darren	Yeah (.) you c I mean you could have an ID generator
		(6.0)
19	Darren	[Yeah]
20	Andy	[OK]
		(2.0)
21	Andy	Ssss let me have a quick think a second
		(1.5)
22	Darren	I think it is going back again it's that (.) reluctant to do something too clever here if this isn't going in the code

23	Andy	I just say do the simplest thing to start with (1.0) do what you said (5.0) *darren typing* and then it's just going to be hmmm how's it going to kno::w that *commenting on d's code*
		(1.0)
24	Darren	It's literally going to create it is ac I mean it is actually going to create it from scratch
25	Andy	So are you thinking
		(10.0)
26	Andy	So that's going to have to [be]
27	Darren	[Well] (.) it doesn't know what the IP is
28	Andy	But how (.) you can't do that can you because that's not going to know
29	Darren	no
30	Andy	That's why we've yeah *laughs* so we either need to (1.0) that shows that's doing too much dun't it
		(6.0)
31	Andy	Can I have a look at that code again
		(18.0)
32	Darren	Terminated the service though
33	Andy	Or you could just huh well no just a session factory or summat that'll do that that'll get away from that problem won't it (2.0) usually have a session factory >but< all you do is give it a session id
34	Darren	OK
35	Andy	And the user ID and that'll return the session XML back and that way we can make an expecta[tion]
36	Darren	[Session] factory and the user factory yeah
		(5.0)
37	Darren	Do you want delivery receipts and stuff like that (.) just send away where no one goes
		(8.0)
38	Andy	I don't know if we need the user one
39	Darren	Potentially because of this bit as well we could there we could if we're gonna do it then we're kind of consistent
40	Andy	Huh let's start with the session one

41	Darren	Yeah (1.0)
42	Andy	Watch it huh oh yeah
43	Darren	I'll start with the session one
44	Andy	yeah
45	Darren	And see how that goes (.) so I'm going to put a mock in *typing*
		(12.0)
46	Andy	It always feel like when you're back at err (2.5) like two s:erver pack service layers for::
47	Darren	That's what I was saying it [does feel]
48	Andy	[Au:thentication] and then for sessions but then (.) we're now replacing them with the factories
49	Darren	I just think this might lead to (0.5)
50	Andy	yeah
51	Darren	Actually having to have
		(10.0)
52	Darren	Public session factory something like that *talking as he codes*
		(35.0)
53	Darren	New card session (3.0) so that way I can create (2.0) create this session then
		(7.0)
54	Darren	Got the session factory set up
		(20.0)
55	Darren	Let the session factory make the decision about the ID
		(2.0)
56	Andy	I think that sounds better as well yeah
		(8.0)
57	Darren	[So that way]
58	Andy	[Do it again]
59	Darren	Then (1.0) the first one is the session ID so that would get an explicit (3.0) so we get that ID
		(2.5)
60	Darren	[there]
61	Andy	[Let]'s do the second one (.) the user name
62	Darren	Yeah

		(3.0)
63	Darren	Then that one's third
64	Andy	Yep
65	Darren	Is that the right I I think user ID should be before session ID
66	Andy	Session ID and session XML should be next to each other yeah
67	Darren	Yeah (1.0) it's kind of doin' it on behalf of
		(1.0)
68	Andy	That's gonna
		(5.0)
69	Andy	Sounds good to me
		Ends at 7'45